Universidad
Carlos III de Madrid

Departamento de Informática

PROYECTO FIN DE CARRERA

Diseño e implementación de un precompilador para permitir herencia múltiple en Java

Autor: Miguel Alonso Luna

Tutor: Manuela Alejandres Sánchez

Director: Gonzalo Génova Fuster

Leganés, enero de 2016

Agradecimientos

Debo agradecer la aportación de Gonzalo Génova y Manuela Alejandres, básica para ser capaz de realizar este proyecto. Igualmente le quiero dar las gracias a mis padres, por su infinita paciencia.

Resumen

La herencia es algo básico en todo desarrollo tecnológico. Sería difícil, por no decir imposible, imaginar un lenguaje orientado a objetos que no la contemplase. Del mismo modo, la herencia múltiple o multiherencia (herencia donde en lugar de un padre existen dos padres o más) es extremadamente útil, pero no está soportado por todos los lenguajes de programación, debido a ciertas dificultades tecnológicas. Java es uno de esos lenguajes.

La idea del proyecto es hacer, de la forma más sencilla posible para el programador, una adaptación del código Java que permita diseñar e implementar el código como si soportase la herencia múltiple, eliminando además las dificultades tecnológicas que esto atañe. Esta aplicación cambiará el código fuente de forma automática y transparente para el programador, de tal manera que se consiga algo tan similar a la herencia múltiple que nos permita utilizarla en diseños y desarrollos.

Java solventa el inconveniente de no tener herencia múltiple a través del uso de interfaces. Combinando la herencia simple y las interfaces se puede uno acercar mucho a la herencia múltiple. Lo que ocurre, es que esto, de cara al usuario que quiere diseñar una aplicación es más complejo, sobre todo conceptualmente, y puede dar lugar a que no se haga bien, o simplemente que no se intente.

La idea de este proyecto es justamente esa, utilizando la potencia de las interfaces que nos permite Java, permitir al usuario diseñar la aplicación usando herencia múltiple y, con este precompilador, pasar dicho diseño de multiherencia a uno compuesto por una herencia simple y un conjunto de interfaces, tal y cómo las buenas prácticas de Java nos instaría a usar.

Abstract

Inheritance is basic to all technological development. It would be difficult, if not impossible, to imagine an object-oriented language that does not allow it. Similarly, multiple inheritance or multi-inheritance (inheritance where instead of a parent are two or more) is extremely useful, but is not supported by all programming languages because of certain technological difficulties. Java is one of those languages.

The project idea is to, in the simplest possible way for the programmer, an adaptation of the Java code that allows design and implement the code as allow multiple inheritance, while eliminating the technological difficulties this regard. This application will change the source code automatically and without effort to the programmer, so that they get something so similar to multiple inheritance to allow us to use it in design and development.

Java solves the inconvenience of not having multiple inheritance using interfaces. Combining single inheritance and interfaces can you get very close to multiple inheritance. What happens is that for the user who wants to design an application this solution is more complex, especially conceptually, and can lead to not do well, or simply not try.

The idea of the project is to use the interfaces that Java let us to allow at the user to design the application using multiple inheritance. With this precompiler we will be able to transform a multi inheritance design to other based in simple inheritance and a set of interfaces, as Java good practices says us that it must be done.

Índice

Índice de ilustraciones

Índice de tablas

1. Introducción

1.1 Descripción breve y objetivos del proyecto

El objetivo del Proyecto Fin de Carrera que se expone en el presente documento es el de permitir la herencia múltiple o multiherencia cuando se realice código en el lenguaje de programación Java.

Actualmente Java no permite realizar la herencia múltiple en su código [1], dando como resultado un error de compilación. Sin embargo otros lenguajes de programación también de gran calado, como puede ser C++ [2] por ejemplo, sí permiten esta alternativa.

El objetivo principal del proyecto será, por tanto, realizar un programa que permita a cualquier desarrollador utilizar la multiherencia de forma sencilla cuando trabaje con Java. Y se quiere no sólo que se pueda utilizar la herencia múltiple, sino que además el programador la realice de manera lo más cómoda posible. Esto es, que programe como si la herencia múltiple estuviese totalmente aceptada por el compilador de Java.

A la hora de afrontar este proyecto es necesario tener en cuenta que en Java se pueden hacer composiciones que emulan la multiherencia, pudiendo ser utilizadas directamente por los programadores. Pero estas composiciones (también llamados patrones) no son en muchos casos obvias, si no hay que conocerlas y aplicarlas. No surgen en el programador de manera natural.

Un patrón será, por tanto, una solución ya definida que soluciona un problema de diseño. Esta solución debe ser efectiva y reutilizable, esto es, se podrá utilizar en más de

un contexto. En el caso que nos atañe el patrón es una combinación de una herencia simple y diferentes interfaces que emularán el funcionamiento de la herencia múltiple.

Además, siempre que se tiende a hacer algo de una manera más compleja el ser humano está más expuesto a errores. Un programador puede conocer el patrón correcto para realizar esta labor, pero al ser una manera mucho más compleja de realizar la posibilidad de fallo humano se incrementa muchísimo. No digamos ya si el programador ni siquiera lo conoce bien.

Se ha seleccionado un patrón, y se pasará a aplicar dicha combinación cada vez que nos encontremos con un caso de multiherencia. Este patrón será explicado en profundidad más adelante en este documento. El objetivo principal de la aplicación será aplicar el patrón mencionado de manera automática.

Existe algún otro patrón posible a utilizar [3], pero el seleccionado es el que mejor cubre las posibles eventualidades que tiene la herencia múltiple, explicadas posteriormente en el documento.

Y, ¿qué significa que el patrón se utilice de manera automática? Lo que viene a decir es que el programador realizará el código en Java con herencia múltiple como si estuviese totalmente permitido su uso por el lenguaje. Y con únicamente un comando de texto, el programa pasará a tener el patrón diseñado en lugar de la multiherencia.

Este programa nuevo tendrá exactamente la misma funcionalidad que el anterior, pero aceptado por el compilador. ¡Será virtualmente igual a estar utilizando herencia múltiple!

Ahí reside una de sus grandes virtudes, el programador apenas notará la diferencia con su programa inicial, luego para él sí será posible utilizar la herencia múltiple en Java.

1.2 Visión general del proyecto

Este documento, en el cual se presenta el Proyecto Fin de Carrera encargado de la aceptación y transformación de la herencia múltiple en Java, se compone de los siguientes capítulos:

- Un primer capítulo de introducción, donde se describe brevemente qué hará el proyecto, así como la definición de los objetivos.

- En el segundo capítulo, estado del arte, se sientan las bases teóricas de aquellos conceptos importantes en relación al proyecto, de manera que el

lector pueda tener unas nociones básicas de aquellos conceptos aquí tratados.

- En el tercer capítulo se pasa a realizar una descripción bastante detallada de qué busca el proyecto. Para ello se definen los posibles usuarios, pero sobre todo, se detalla los distintos requisitos de la aplicación.

- Posteriormente, en el cuarto punto, se realiza un diseño detallado de la aplicación, fundamentalmente a nivel técnico. Se detallará con un diagrama de Casos de Uso y con diagramas de Clases.

- El quinto punto corresponde a una explicación detallada del código. En él se comentarán, brevemente, los puntos más relevantes del código de la aplicación.

- El siguiente punto, el sexto, hace referencia a las distintas pruebas realizadas en el programa. En ellas se comentan tanto las pruebas estáticas como las dinámicas. Además, se desarrolla un poco más una prueba del sistema, que incluirá varios tipos de pruebas.

- El punto séptimo trata distintos datos asociados al proyecto que no fueron recogidos en puntos previos. Trata fundamentalmente la planificación, como las distintas herramientas utilizadas para desarrollar este programa.

- El punto octavo hace referencia al presupuesto teórico del programa. En él se detallan los distintos costes, tales como el de personal, el de hardware, software, etc. Así como un resumen final incluyendo todos.

- En el último punto, el noveno, se pasan a describir las conclusiones del autor, así como los principales problemas encontrados para el desarrollo de la aplicación. Además se añade un punto en el que se discute las posibles evoluciones que este programa podría tener en el futuro.

2. Descripción general

Este capítulo describe el contexto en el que debe encuadrarse el producto final. Está dividido en los siguientes cinco puntos.

2.1 Capacidades generales

Este programa se trata de un producto novedoso, esto es, no se conoce ninguna aplicación (al menos que esté disponible de forma abierta) que ofrezca las funcionalidades que este aplicativo da.

La principal meta del aplicativo es ayudar al programador o analista que utilice Java como lenguaje de desarrollo.

La ayuda que ofrece éste aplicativo es sobre un punto muy concreto, pero que puede ser de gran utilidad, ya que permite utilizar en Java la herencia múltiple. No siempre un programador usará esta característica, pero este aplicativo permite su uso en los casos en los que el programador (o persona que diseñe la aplicación) crea que es conveniente. En estos casos la ayuda que se reciba puede ser enorme, dando lugar a un importante ahorro en tiempo y en costes.

La principal tarea del programa es permitir la programación en Java utilizando multiherencia. Por tanto el programador, analista, arquitecto, etc. podrá programar como si esta característica estuviese recogida en Java, y tras usar esta aplicación, sin coste en tiempo para el usuario, obtener la conversión de algo que Java no permite (herencia múltiple) a algo que Java sí acepta (utilizará un algoritmo o patrón) para la conversión.

Por consiguiente, el objetivo del programa está claro. Dada una entrada no permitida por Java la aplicación deberá devolver una salida que el compilador de Java sí acepte. Ésta salida tendrá un formato similar a la entrada y, por supuesto, una funcionalidad similar.

El programa será capaz de operar todos los puntos que la herencia múltiple permite. Esto es, será capaz de distribuir tanto las operaciones como los atributos, así como otras posibles referencias entre clases. Este proceso es costoso y complejo. En siguientes puntos de este documento será explicado en profundidad.

2.2 Restricciones generales

Esta aplicación, que acepta prácticamente cualquier tipo de herencia múltiple, tiene aun así algunas restricciones.

La principal es que el lenguaje de programación en el programa de entrada que se convertirá debe ser Java. No aceptará ningún otro tipo de lenguaje de programación para crear el programa que se toma como entrada. Bueno, realmente lo aceptará, pero simplemente creará una copia exacta de él en el fichero de salida, sin ningún tipo de cambio en la multiherencia. Será, pues, un paso sin sentido.

Además de tener que ser Java el programa de entrada, deberá ser una versión 1.6 como máximo. Para versiones posteriores su buen funcionamiento no estará asegurado, aunque para un grandísimo porcentaje de casos sí funcionará en versiones más recientes. Su adaptación a versiones más modernas de Java es un buen trabajo de mantenimiento y ampliación de este programa.

El programa de entrada, además de código fuente, puede tener otros ficheros tales como documentos, fotografías, vídeos, etc. Este programa puede tener problemas para hacer copias de diversos ficheros, fundamentalmente aquellos que no sean de texto. Esto se hace especialmente relevante para aquellos ficheros que tengan un tamaño elevado. Para vídeos, por ejemplo, es mejor opción quitarlos del programa de entrada, ya que seguramente su copia no sea posible y copiarlos a mano en el de salida, en caso de ser necesario.

Java permite la escritura del código de diversas maneras. Permite, por ejemplo, escribir cada palabra en una línea diferente, dejando incluso varias líneas en blanco entre palabra y palabra. Obviamente nadie lo escribe así, pero en caso de que alguien decidiese hacerlo, ya que es posible, posiblemente daría problemas. El programa responde bien ante el hecho de que se pueden escribir las cosas de distintas formas, y ante las variantes más comunes no tiene problema. Pero podría fallar ante algunas formas, raras por lo general, de escribir las cosas. Se entiende que en un entorno normal, bien académico bien laboral, nunca se escribirá el código de una forma que haga fallar al precompilador, pero si se busca la manera de hacerlo fallar a propósito posiblemente se podría conseguir. Aunque esto implicaría que el código escrito sea difícilmente legible y reutilizable.

Además, si el programa Java de entrada tiene errores, bien de compilación o bien de ejecución, antes de su conversión, y estos problemas no están causados por la herencia múltiple, esos mismos errores seguirán persistiendo en la salida.

2.3 Características de los usuarios: roles y capacidades

Existirán tres tipos de usuarios de esta aplicación: los usuarios expertos, los usuarios ocasionales y el personal de mantenimiento.

- Los **usuarios expertos** serán el rol más habitual que utilice la aplicación. Una estimación aproximada dice que dos de cada tres personas que la utilicen serán usuarios expertos.
  Se trata de una persona con un fuerte conocimiento de programación, más concretamente del lenguaje Java. Por lo general la experiencia de este tipo de usuarios con otras herramientas informáticas de creación o modificación de código, tales como las herramientas case, será elevada.
  Es, por tanto, un grupo con una curva de aprendizaje respecto a este tipo de herramientas muy corta. En breve tiempo sabrán sacarle todo el partido.

- Existe otro tipo de usuario, llamado **usuario ocasional**. Se trata de gente que utilice la aplicación para ver únicamente la evolución de las clases, aunque no tiene por qué programar en lenguaje Java. Esto es, si tú tienes un problema similar con la herencia múltiple con el lenguaje de programación que utilizas similar al que Java tiene, puedes utilizar esta herramienta a modo de consulta, para comprobar cómo lo hace, aunque el código generado no te sea tan útil a como lo sería si usases Java.
  Este tipo de usuario también es una persona muy técnica, aunque no por ello deba ser una experta en Java. Pero desde luego deberá ser capaz de programar en algún otro lenguaje, o si no, al menos, será capaz de conocer el lenguaje de modelado UML.

- El tercer y último tipo de rol es el de **personal de mantenimiento**. Este perfil será en el que el futuro siga haciendo crecer esta aplicación, bien integrándola con otras, bien añadiéndole nuevas características o bien corrigiendo algún pequeño bug que pudiesen encontrar.
  De nuevo se trata de un perfil muy técnico. Será un conocedor experimentado de los lenguajes de programación en general y de Java en particular.
  Necesitará, eso sí, para la realización de su labor, a diferencia de los otros dos roles mencionados anteriormente, el código fuente de la aplicación.

Como se puede observar no existe el perfil administrador. Esto es debido a que todo usuario puedo utilizar el total de las funcionalidades de la aplicación.

De este estudio de los roles se puede sacar la conclusión que todos son, en mayor o menor medida, gente de un perfil altamente técnico. Luego el manejo de esta aplicación, que además en bastante sencillo en su uso, no así en su algoritmia interna, no debería suponerle un gran problema a estos perfiles, ya que se les intuye cierta familiaridad con sistemas similares.

Además, se obtiene la conclusión que no es necesaria una gran interfaz de usuario, que incluso una interfaz basada en comandos sería más que suficiente, debido a lo técnico de los perfiles que la utilizarán.

3. Estado del arte

Aunque las tecnologías y distintos conceptos tecnológicos usados en este proyecto son muy populares, y por tanto, ampliamente conocidos, se explicarán de manera muy breve los principales conceptos tecnológicos.

3.1 Java

Java es un lenguaje de programación orientado a objetos, que desarrolló Sun Microsystems. Actualmente Java pertenece a Oracle, tras la compra de Sun por esta compañía en el año 2009.

Java tiene entre sus principales características la de ser un lenguaje compilado e interpretado. Para ello cada programa realizado con dicho lenguaje deberá compilarse, generándose un código que será interpretado posteriormente por una máquina virtual

De esta manera se consigue una de sus principales ventajas, que consiste en que Java es independiente de la plataforma. Esto significa que haciendo un único programa en Java éste funcionará en cualquier ordenador, independientemente del sistema operativo que use. Además funcionará en otros dispositivos, con la única obligación de que tengan la máquina virtual. Esto supone una gran virtud, porque así se evita tener que hacer varios desarrollos iguales.

Respecto a su sintaxis, ésta es muy parecida a la de C++. Pero Java no es una evolución de dicho lenguaje, es totalmente independiente.

Java dispone de un conjunto de herramientas de desarrollo. Éstas se conocen como Java Development Kit(JDK). Se pueden descargar, de manera gratuita, en la siguiente página: http://java.sun.com.

3.2 Compilador

En esencia un compilador es una herramienta o programa que toma una entrada, que sería el lenguaje de alto nivel dado por el programador, Java en este caso, y lo transforma en un lenguaje que el ordenador puede entender, el lenguaje máquina (bytecodes).

Ilustración 1: Compilador [4]

Este tratamiento tan sencillo descrito arriba se suele clasificar, de manera teórica, en distintas fases. Estas fases se muestran en la siguiente imagen, aunque en la práctica los compiladores suelen agrupar algunas:

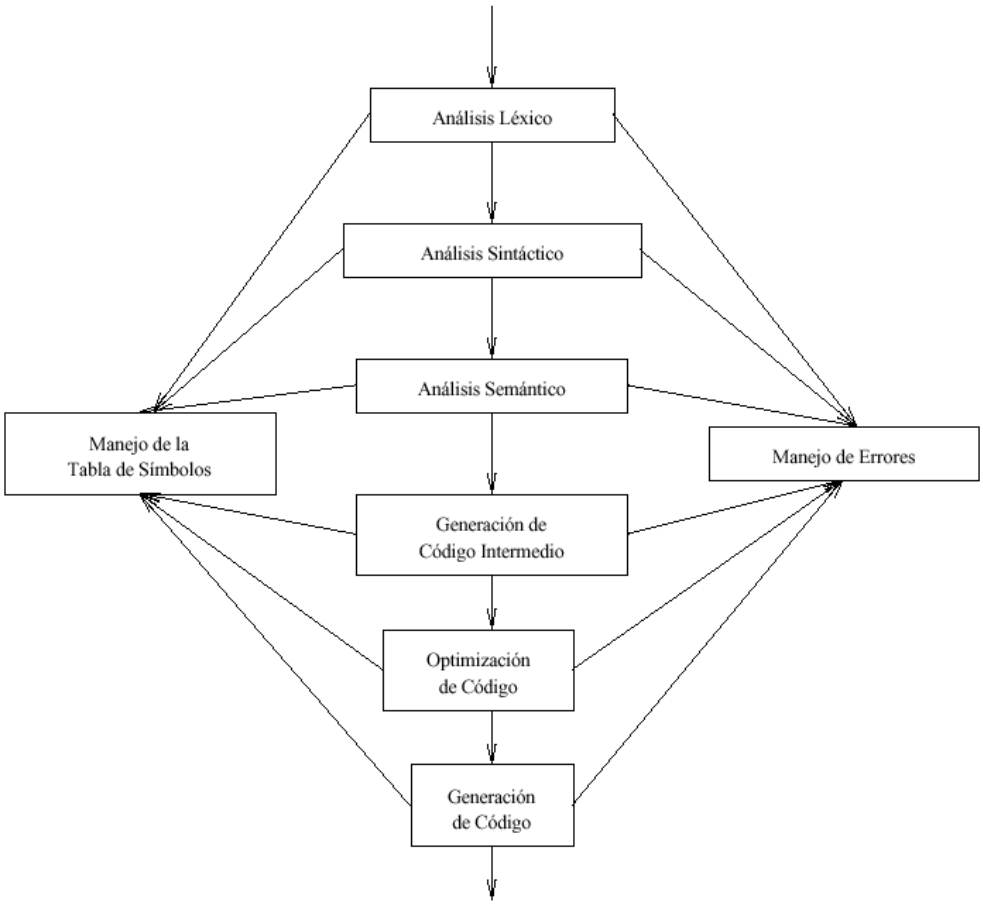

Ilustración 2: Fases de un compilador [5]

Respecto al compilador de Java, debemos ejecutarlo cada vez que cambios el código. La invocación se hace a través del comando "javac". Pasaremos de esta forma de tener una clase con extensión .java a otra, la que realmente leerá el ordenador, del tipo .class.

3.3 Herencia en Java

La herencia es una característica común a todos los lenguajes de programación orientados a objetos. En Java es enormemente utilizada.

¿Por qué se debe usar la herencia? La herencia debe usarse siempre que al necesitar crear una nueva clase y se vea que ya existe otra ya desarrollada con suficientes similitudes. De esta manera se adoptarán características ya implementadas y probadas, suponiendo un ahorro muy importante de tiempo y esfuerzo.

La estructura jerárquica de la herencia es la siguiente: una clase tiene únicamente una clase padre, también conocida como superclase. Al mismo tiempo, la superclase puede tener cualquier número de clases hijas, llamadas también subclases.

En el siguiente ejemplo:

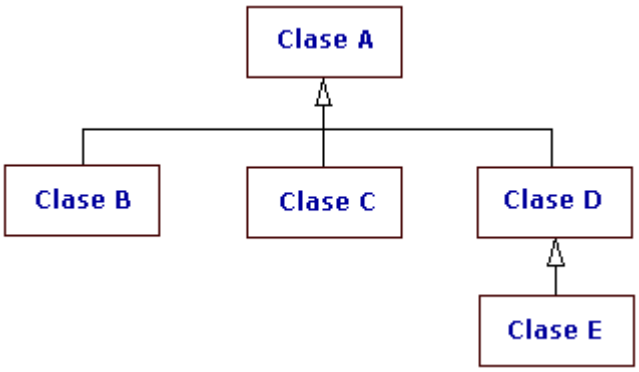

Ilustración 3: Herencia simple [6]

- A es la superclase de B, C y D.
- D es la superclase de E.
- B, C y D son subclases de A.
- E es una subclase de D.

Esto afectará a la subclase de la siguiente manera:

- Pasará a heredar todas las características de la superclase.
- La subclase podrá definir características adicionales.
- La subclase será capaz de redefinir las características que ha heredado..
- El proceso de herencia afectará únicamente a la clase hija, nunca a la superclase existente.

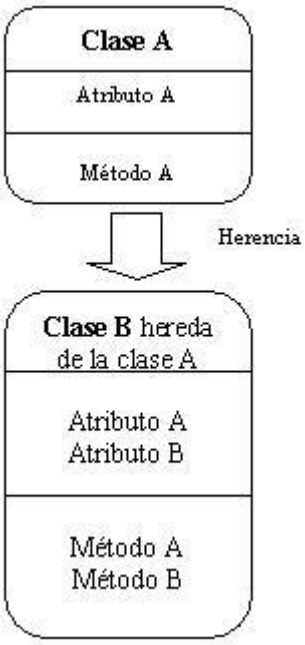

Ilustración 4: Atributos y métodos en herencia [7]

Respecto a la implementación: para conseguir que haya herencia basta con añadir la palabra reservada extends y a continuación el nombre de la clase padre después del nombre de la clase nueva:

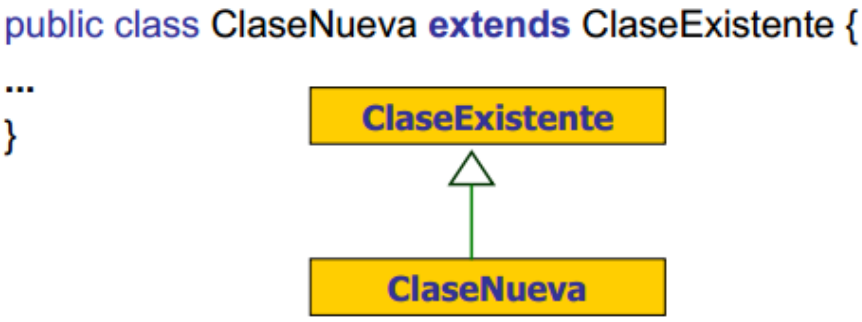

Ilustración 5: Extends

A continuación se muestra un ejemplo sencillo con tres clases, la superclase Persona, y su subclase o clase hija Empleado. A su vez, esta subclase será superclase de otra, Directivo. Sabemos, pues, que Directivo tendrá todas las características (atributos y métodos) de la clase Persona.

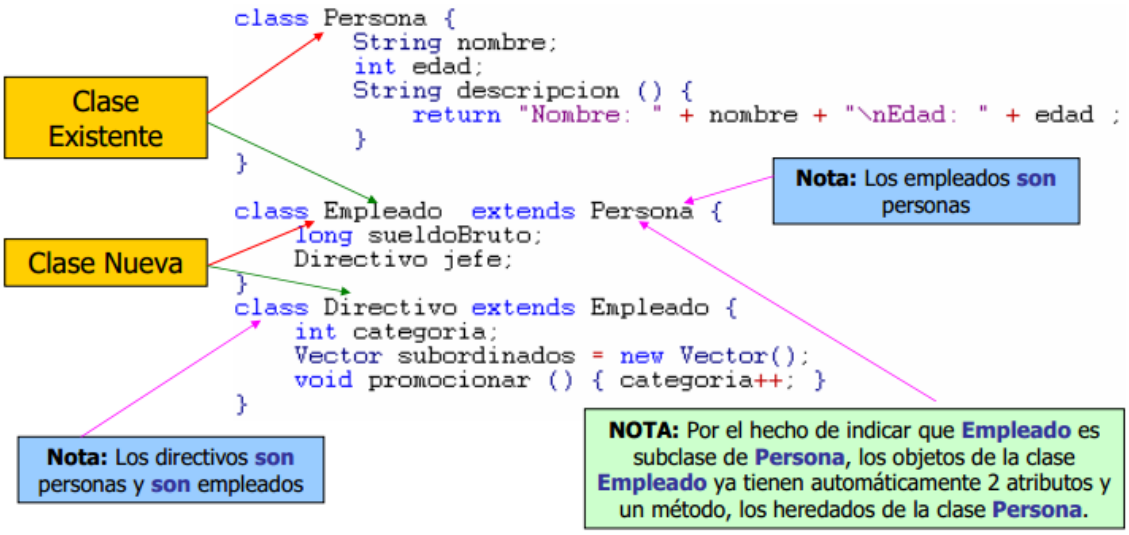

Ilustración 6: Código herencia

3.4 Interfaces en Java

El concepto de interfaz en Java hace referencia a un tipo de clase donde se especifica qué se debe hacer, pero no incluye una implementación de esta funcionalidad. Por tanto, se necesitará que haya otras clases que implementen dichas interfaces.

De esta manera tienes una forma de asegurarte que una clase cumple una serie de requisitos al implementar una interfaz concreta. Además te permite conectar de alguna manera clases que a priori no tienen ningún nexo entre ellas, y además obliga a este grupo de clases que, por ejemplo, implementan un método similar, a que todas tengan dicho método con exactamente el mismo nombre.

Un ejemplo sencillo de código de cómo crear una interfaz podría ser el siguiente:

```
public interface VideoClip {
    // comienza la reproduccion del video
    void play();
    // reproduce el clip en un bucle
    void bucle();
    // detiene la reproduccion
    void stop();
}
```

Ilustración 7: Ejemplo código interfaz [8]

Las clases que quieran utilizar la interfaz VideoClip utilizarán la palabra reservada implements y deberán proporcionar el código necesario para implementar los métodos que se han definido para la interfaz:

```
class MiClase implements VideoClip {
    void play() {
        <código>
        }
    void bucle() {
        <código>
        }
    void stop() {
        <código>
    }

}
```

Ilustración 8: Ejemplo implementación interfaz [8]

Una interfaz puede ser implementada por varias clases al mismo tiempo, al igual que una clase puede implementar varias interfaces a la vez. No hay ningún tipo de restricción en Java en este aspecto, como sí lo hay con la herencia:

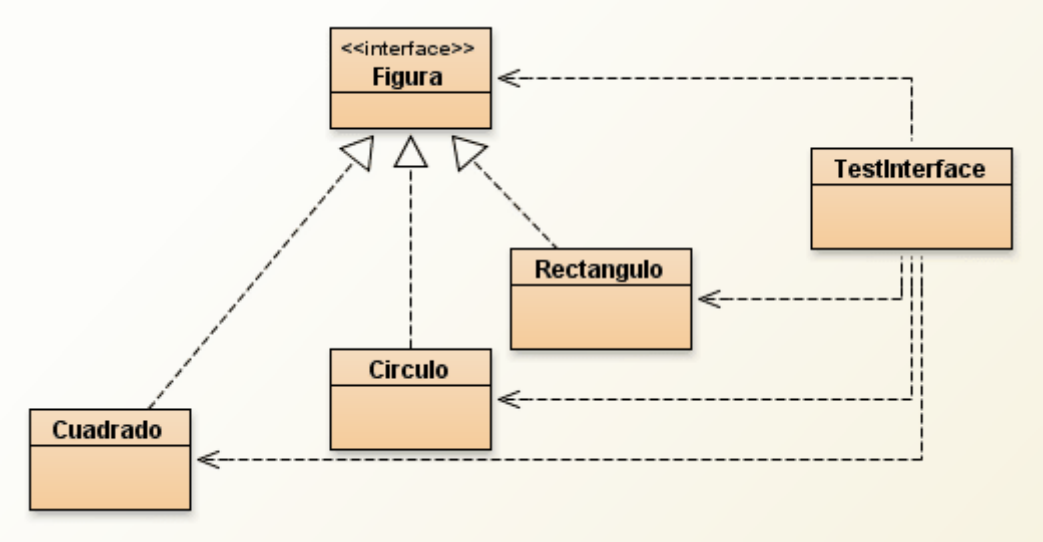

Ilustración 9: Interfaces Java [9]

3.5 Herencia múltiple

La herencia múltiple es una característica similar a la herencia simple explicada anteriormente, pero con la salvedad que en este caso permite que existan varias superclases para una subclase, o lo que es lo mismo, varios padres para un mismo hijo. Por ejemplo:

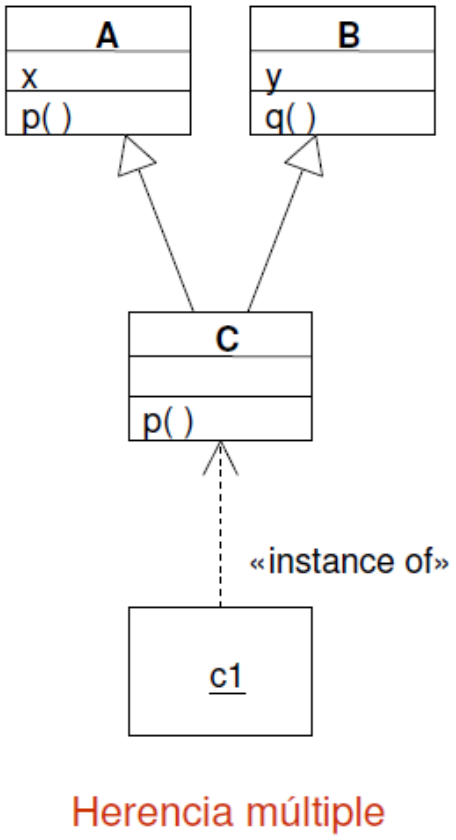

Ilustración 10: Definición herencia múltiple [10]

La herencia múltiple no es soportada por Java, únicamente lo es la herencia simple. Otros lenguajes que sí soportan herencia múltiple son: Phyton, C++, Eiffel, Perl, CLOS, Object REXX y Centura SQL Windows.

Esto es debido a que la herencia múltiple tiene una serie de contras, que se explican más detalladamente a continuación, que hace que ciertos lenguajes prefieran no permitirla si tienen mecanismos con los que puedan mitigar su ausencia.

A través de las interfaces y de la delegación Java consigue algo muy cercano a la herencia múltiple. Este es un recurso que muchos otros lenguajes no poseen, y por el cual Java decidió prescindir de la herencia múltiple, eliminando los problemas que ello

conlleva pero perdiendo por otro lado la enorme potencia que la herencia múltiple permite en cuanto a facilidad de diseño y reutilización (punto siguiente).

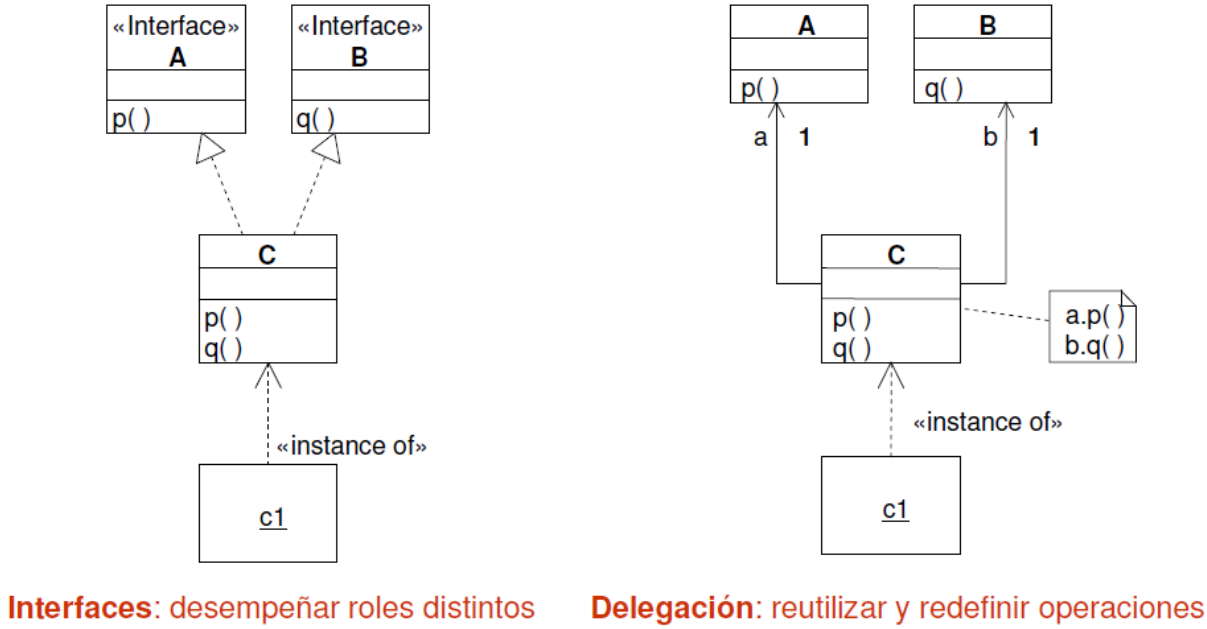

Ilustración 11: Herencia múltiple [11]

Problemas de la herencia múltiple: Problema del diamante

En los lenguajes de programación orientada a objetos que implementan la herencia múltiple, el problema del diamante es una ambigüedad de muy difícil solución que surge cuando dos clases, llamadas en este caso B y C, heredan de A, y la clase D hereda de B y C. Si un método en D llama a un método definido en A, ¿por qué clase lo hereda, B o C?

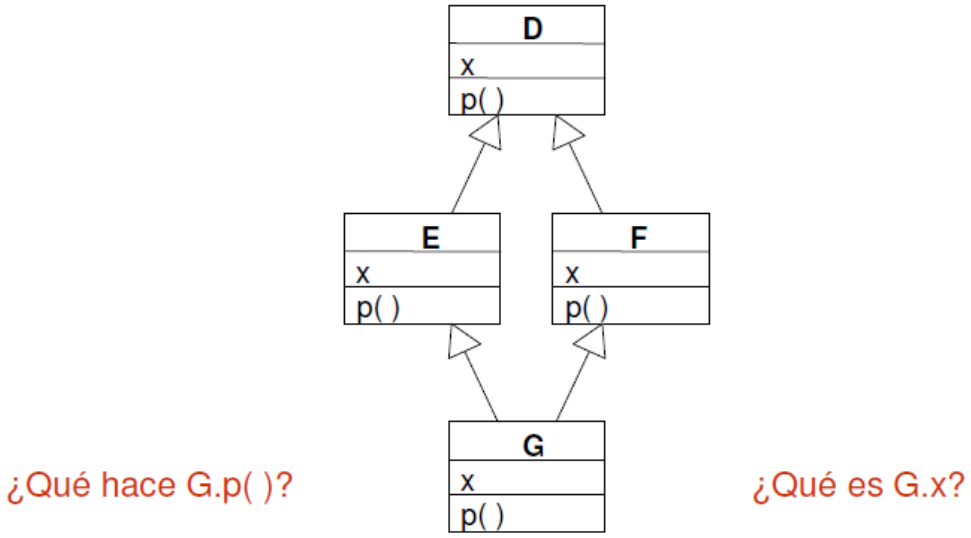

Ilustración 12: Problema del diamante [12]

En este ejemplo, se da además un problema llamado "colisión de nombres" [13]. Cuando la clase G llama al método p(), ¿a qué clase de las dos padres está invocando? Existe el mismo problema para el atributo x.

3.6 Ventajas de la herencia múltiple

El uso de la herencia múltiple tiene varias ventajas [14], aunque en este punto se hará más hincapié en las dos ventajas principales que nos permite la multiherencia, como son una mayor adaptación semántica y una migración de lenguaje más sencilla.

Mayor precisión en la definición

Como se comentó previamente, la herencia se debería usar siempre que se encuentre similitudes entre una clase nueva que se quiera crear y otra clase que ya haya sido desarrollada previamente.. De esta manera, se adoptarán características ya implementadas y probadas, suponiendo un ahorro muy importante de tiempo y esfuerzo. [15]

Y es que uno de los principales objetivos de la programación orientada a objetos es el incremento de la productividad en el desarrollo del software. Y una de las principales maneras de conseguirlo es a través de la reutilización de código existente, donde las relaciones de herencia forman uno de los principales medios para conseguirlo. [16]

Habrá casos en los que se podrá definir la situación real con una sola clase padre. Pero también encontraremos otros donde esto no sea posible, y debamos tener de manera obligatoria varios padres. Para ello necesitaríamos herencia múltiple.

Un ejemplo clásico es la clase "Anfibio". Esta clase debería heredar, si estuviésemos hablando de medios de transporte, por ejemplo, de las clases "Barco" y "Automóvil", para tomar las características que queremos de cada uno de ellos, ya sea desplazarse por carretera del primero o desplazarse por el mar, por ejemplo, del otro. [17]

Con herencia simple esto no es posible. Tan sólo se podría tener un padre, y deberíamos repetir características ya diseñadas y probadas, con el incremento de tiempo y coste que ello conlleva.

Por supuesto, se podría mitigar este defecto de Java haciendo un muy buen diseño, a través de interfaces y clases intermedias. De hecho es lo que la aplicación hace, en esencia. Pero a nivel de diseño, hacerlo de esta manera es mucho más difícil, es infinitamente más intuitivo usar la multiherencia.

De hecho, se puede afirmar que en un gran número de casos que el diseño se haría bien con herencia múltiple, al no disponer de ella, aunque haya métodos alternativos, no se hará correctamente, al ser mucho más alta su dificultad.

Reutilización Diseño

Cuando se realiza el diseño de una aplicación, hay partes del diseño que deberían ser independientes del lenguaje de programación.

De hecho UML, que es el lenguaje con el que habitualmente se lleva a cabo el diseño técnico de las aplicaciones, es independiente del lenguaje con el que finalmente se implemente dicha aplicación.

Ciertamente, aunque la teoría y las buenas prácticas nos indican esa independencia del lenguaje, en la vida real pocas veces se hace así. Pero es cierto que el hecho de que en ciertos lenguajes se tenga herencia múltiple y en otros no (como en Java) es un punto negro en este apartado.

Pero el problema grande viene en el aspecto de las migraciones (también llamadas en estos casos actualizaciones). Java es un lenguaje de moda en la actualidad, y muchas aplicaciones de lenguajes más "viejos" se están pasando a este lenguaje. Java es el lenguaje de programación más utilizado en el mundo actualmente [18], y el que más migraciones realiza. Esto es, se están actualizando aplicaciones antiguas a nuevas aplicaciones realizadas en Java.

Y varios de estos lenguajes más "viejos" sí permiten herencia múltiple. ¿Qué significa esto?, pues que pasamos de un lenguaje con multiherencia a uno que no la tiene. Como consecuencia tendríamos que no nos valdría el diseño de la primera aplicación, ya que podría contener herencia múltiple.

Esto complicaría la migración, desde luego. Deberías cambiar más cosas, ¡cambiar incluso el diseño! El hecho de poder tener herencia múltiple en Java y de esta manera reutilizar el diseño previo sería una gran ventaja, desde luego.

4. Análisis

4.1 Análisis de Requisitos

En este apartado se pasa a definir de manera detallada los requisitos software del sistema, requisitos que definirán qué hará la aplicación desarrollada posteriormente.

4.2 Especificación de Requisitos

Hemos dividido los requisitos en los siguientes tipos:

- Funcionales: define las funciones que debe realizar el software.
- De rendimiento: tiempo que debe tardar como máximo la aplicación en diferentes escenarios.
- De interfaz: referentes a la interfaz gráfica de usuario.
- De recursos: software necesario. También se podría mencionar el hardware en caso de necesitarse algo muy concreto.
- De comprobación: comprobaciones necesarias a hacer antes de ejecutar la aplicación. Generalmente en el ámbito de las conexiones.
- De documentación: define si es necesaria alguna documentación.
- De seguridad: se dice cómo tratar información sensible en caso de haberla, como las contraseñas por ejemplo.
- De daño: define el comportamiento ante fallos.

A continuación se procederá a explicar brevemente el formato que ha sido utilizado para la captura de requisitos:

Identificador		**Tipo**	
Descripción Breve			
Descripción Detallada			
Pruebas a realizar			
Fuente		**Necesidad**	

Tabla 1: Formato estándar de requisitos

- **Identificador**: identificará unívocamente los requisitos, facilitando su trazabilidad. El formato que utilizará es el siguiente:

  RS-[XXX]

  Donde:

  RS: define que se trata de un requisito de software.
  XXX: número de requisito que se asigna de manera secuencial, empezando por 001.

- **Tipo:** se trata del tipo de requisito, que en nuestros casos podrá tomar los siguientes valores:

  - FUNC: requisito de tipo funcionalidad. Explica la funcionalidad que debe tener la aplicación, esto es, qué debe hacer.
  - REND: requisito de tipo rendimiento. Se parametrizarán ciertas variables concernientes al rendimiento de la aplicación, mediante la asignación de valores numéricos.
  - INTF: requisito de tipo interfaz. Hará referencia a la interfaz gráfica del programa.
  - REC: requisito de tipo recursos. Detallarán los recursos necesarios a nivel de hardware.
  - COMP: requisito de tipo comprobación. Se checkearán las entradas y salidas del sistema.
  - DOC: requisito de tipo documentación. Detallará como debe realizarse la documentación del proyecto.
  - SEG: requisito de tipo seguridad. Detallará las pautas referentes a la seguridad del programa.

- DAM: requisito de tipo daño. Detalla cualquier requisito que pueda dar lugar el fracaso del software, para reducir el posible daño.

- **Descripción breve:** pequeña definición de lo que pretende el requisito.
- **Descripción detallada:** ampliación de la descripción breve, en caso de ser necesario.
- **Pruebas a realizar:** pequeña definición de en qué consisten las pruebas que deben realizarse para testear el requisito.
- **Fuente:** origen del requisito.
- **Necesidad:** determina el grado de importancia que se le otorga a cada requisito. Los valores que puede adoptar son los siguientes:

  - Esencial: máxima prioridad. Deberán implementarse de manera obligatoria.
  - Deseable: requisitos todavía con una importancia bastante alta, pero ya pudiéndose plantear su ausencia.
  - Opcional: mínimo grado de importancia, para requisitos poco importantes.

4.2.1 Requisitos funcionales

Identificador	RS-001	**Tipo**	FUNC
Descripción Breve	Selección entrada		
Descripción Detallada	El usuario podrá seleccionar la entrada, a través de línea de comandos.		
Pruebas a realizar	Se comprobará que exista la posibilidad de introducir la ruta de entrada. Para ello se introducirá una entrada a través de la línea de comandos.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 2: Requisito funcional RS-001

Identificador	RS-002	**Tipo**	FUNC
Descripción Breve	Selección salida		
Descripción Detallada	El usuario podrá seleccionar la carpeta en la que se pondrá los ficheros resultantes de haber ejecutado la aplicación.		
Pruebas a realizar	Se comprobará que exista la posibilidad de introducir la ruta de salida. Para ello se introducirá una salida a través de la línea de comandos.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 3: Requisito funcional RS-002

Identificador	RS-003	**Tipo**	FUNC
Descripción Breve	Error selección de entrada		
Descripción Detallada	El programa informará los errores ocurridos en caso de una mala selección de la carpeta de entrada a través de un mensaje.		
Pruebas a realizar	Se introducirá una carpeta de entrada incorrecta para comprobar si la aplicación avisa tal eventualidad con un mensaje.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 4: Requisito funcional RS-003

Identificador	RS-004	**Tipo**	FUNC
Descripción Breve	Error selección de salida		
Descripción Detallada	El programa informará los errores ocurridos en caso de una mala selección de la carpeta de salida a través de un mensaje.		
Pruebas a realizar	Se introducirá una carpeta de salida incorrecta para comprobar si la aplicación avisa tal eventualidad con un mensaje.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 5: Requisito funcional RS-004

Identificador	RS-005	**Tipo**	FUNC
Descripción Breve	Modificación sufijos		
Descripción Detallada	El cliente deberá ser capaz de modificar el sufijo que llevarán las clases nuevas.		
Pruebas a realizar	Se modificarán los sufijos.		
Fuente	Cliente	**Necesidad**	Opcional

Tabla 6: Requisito funcional RS-005

Identificador	RS-006	**Tipo**	FUNC
Descripción Breve	Sufijo por defecto		
Descripción Detallada	La aplicación pondrá un sufijo por defecto en caso de no ser modificado previamente el sufijo por parte del usuario.		
Pruebas a realizar	Se dejará el sufijo por defecto para comprobar su buen funcionamiento.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 7: Requisito funcional RS-006

Identificador	RS-007	**Tipo**	FUNC
Descripción Breve	Errores de compilación		
Descripción Detallada	El programa generará unas clases de salida, cuyo código no contendrá ningún error de compilación relacionado con multiherencia		
Pruebas a realizar	Se comprobará el código de salida, compilando el código generado con un jdk 1.6, para comprobar que efectivamente no de errores de compilación.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 8: Requisito funcional RS-007

Identificador	RS-008	**Tipo**	FUNC
Descripción Breve	Búsqueda de multiherencia		
Descripción Detallada	El programa buscará y discriminará las clases afectadas por la herencia múltiple en la carpeta seleccionada como entrada.		
Pruebas a realizar	Se comprobará que el programa no deje ninguna relación de herencia múltiple sin seleccionar. Para ello se ejecutará la aplicación sobre un programa con varios casos de herencia múltiple, comprobando que todos ellos han sido atendidos.		
Fuente	Cliente	**Necesidad**	Deseable

Tabla 9: Requisito funcional RS-008

Identificador	RS-009	**Tipo**	FUNC
Descripción Breve	Clases no java		
Descripción Detallada	La aplicación copiará las clases que no sean Java en el directorio de salida sin ningún tipo de modificación.		
Pruebas a realizar	Se comprobará que las clases que no sean Java han sido copiadas al directorio de salida sin ser modificadas. Para ello se ejecutará la aplicación y se comprarán los ficheros no java del programa antes y después de tratar, para comprobar si realmente son iguales en este aspecto.		
Fuente	Cliente	**Necesidad**	Deseable

Tabla 10: Requisito funcional RS-009

Identificador	RS-010	**Tipo**	FUNC
Descripción Breve	Clases java no afectadas		
Descripción Detallada	La aplicación copiará las clases Java que no se hayan visto afectadas por la multiherencia en el fichero que le corresponda de la salida sin ninguna modificación.		
Pruebas a realizar	Se comprobará que las clases Java que no se hayan visto afectadas por la multiherencia han sido copiadas al directorio de salida sin ser modificadas.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 11: Requisito funcional RS-010

4.2.2 Requisitos de rendimiento

Identificador	RS-011	**Tipo**	REND
Descripción Breve	Tiempo ejecución proyectos pequeños		
Descripción Detallada	El programa ejecutará aquellos proyectos con tamaño menor a 100MB en un tiempo inferior a 90 segundos.		
Pruebas a realizar	Se comprobará que los programas con un tamaño máximo de 100 MB son ejecutados en menos de 90 segundos.		
Fuente	Cliente	**Necesidad**	Deseable

Tabla 12: Requisito de rendimiento RS-011

Identificador	RS-012	**Tipo**	REND
Descripción Breve	Tiempo ejecución proyectos grandes		
Descripción Detallada	El programa ejecutará aquellos proyectos con tamaño entre 100 y 600MB en un tiempo inferior a 240 segundos.		
Pruebas a realizar	Se comprobará que los programas con un tamaño comprendido entre 100MB y 600MB son ejecutados en menos de 240 segundos.		
Fuente	Cliente	**Necesidad**	Deseable

Tabla 13: Requisito de rendimiento RS-012

4.2.3 Requisitos de interfaz

Identificador	RS-013	**Tipo**	INTF
Descripción Breve	Definición interfaz		
Descripción Detallada	El programa tendrá una interfaz de comandos, que pedirá la ubicación del fichero o carpeta de entrada y solicitará que le digamos en qué carpeta querremos la salida.		
Pruebas a realizar	Se comprobará que al ejecutar la aplicación la interfaz solicite el fichero de entrada y la ruta para el de salida.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 14: Requisito de interfaz RS-013

4.2.4 Requisitos de recursos

Identificador	RS-014	**Tipo**	REC
Descripción Breve	Software necesario		
Descripción Detallada	El ordenador deberá tener instalado el siguiente software: • Java Runtime Enviroment. Necesaria al menos la versión 1.6.		
Pruebas a realizar	N/A.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 15: Requisito de recursos RS-014

Identificador	RS-015	**Tipo**	REC
Descripción Breve	Software recomendado		
Descripción Detallada	Un visor de texto plano es necesario para poder visualizar los archivos Java.		
Pruebas a realizar	N/A.		
Fuente	Cliente	**Necesidad**	Opcional

Tabla 16: Requisito de recursos RS-015

4.2.5 Requisitos de documentación

Identificador	RS-016	**Tipo**	DOC
Descripción Breve	Manual de usuario		
Descripción Detallada	Se les facilitará a los usuarios un pequeño manual de usuario en el idioma español, donde se explicará el funcionamiento de la aplicación.		
Pruebas a realizar	N/A.		
Fuente	Cliente	**Necesidad**	Deseable

Tabla 17: Requisito de documentación RS-016

4.2.6 Requisitos de daño

Identificador	RS-017	**Tipo**	DAM
Descripción Breve	Robusto ante fallos de la interfaz de usuario		
Descripción Detallada	Se garantizará que el sistema sea robusto ante fallos del usuario introducidos a través de la interfaz		
Pruebas a realizar	Se comprobará el funcionamiento del programa cuando se introduzcan fallos en la interfaz de usuario. Para ello se ejecutará la aplicación introduciendo diversos fallos como entrada de manera premeditada, comprobando que el programa sea robusto ante éstos. Los distintos fallos a introducir son: selección de un fichero de entrada vacío, selección de un fichero de salida con mal formato, etc.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 18: Requisito de daño RS-017

Identificador	RS-018	**Tipo**	DAM
Descripción Breve	Robusto ante fallos		
Descripción Detallada	Se garantizará que el sistema sea robusto frente a ficheros de entrada mal formados.		
Pruebas a realizar	Se comprobará el funcionamiento del programa ante ficheros de entrada mal formados.		
Fuente	Cliente	**Necesidad**	Esencial

Tabla 19: Requisito de daño RS-018

5. Diseño Detallado

5.1 Diagramas de Casos de Uso

A continuación se muestra el diagrama de Casos de Uso del sistema.

Como se puede observar, el diagrama de Casos de Uso de la aplicación es sencillo, ya que la funcionalidad básica del sistema es una, la conversión de la herencia múltiple en herencia simple.

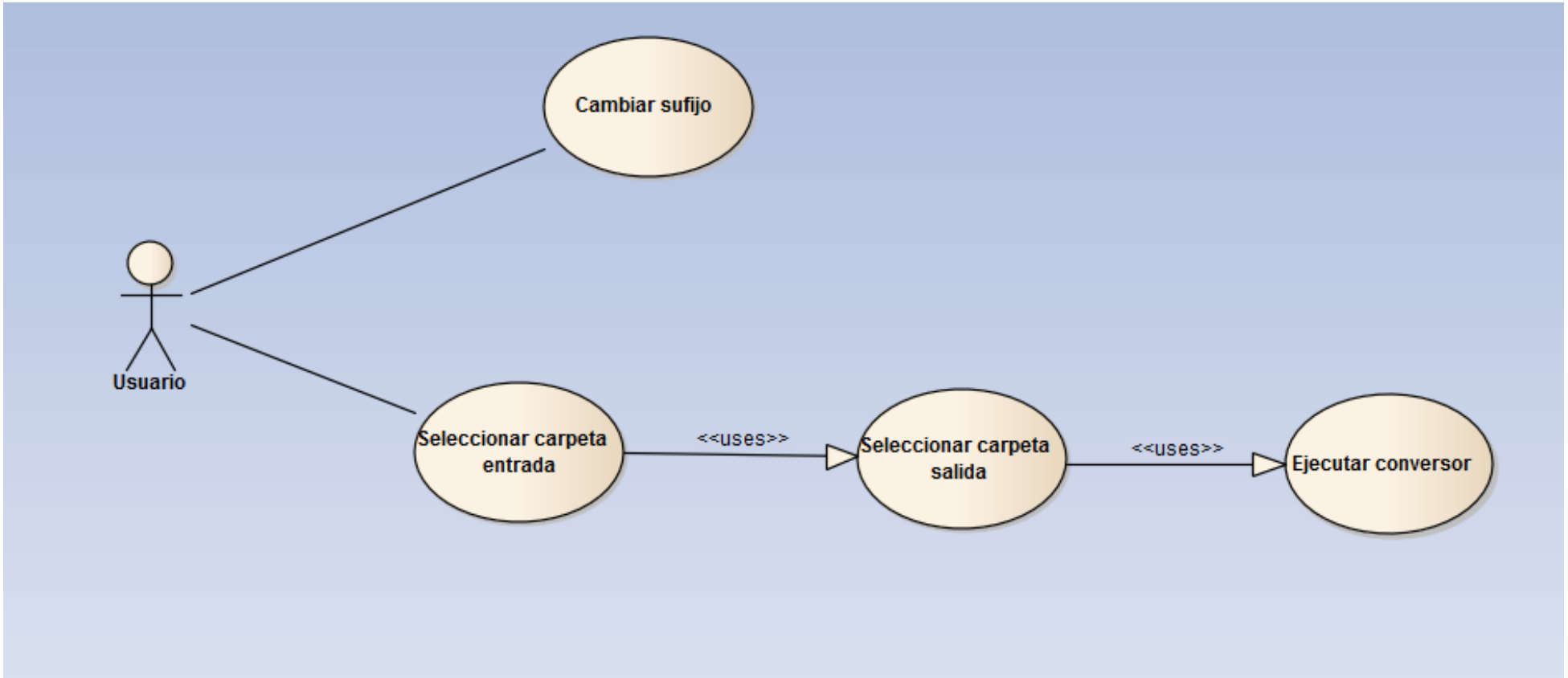

Ilustración 13: Diagrama de Casos de Uso

5.1.1 Descripción textual de Casos de Uso del Usuario

Nombre	**Cambiar sufijo**
Actores	Usuario
Objetivo	El usuario elige el sufijo que llevarán las clases
Precondiciones	El usuario habrá ejecutado la aplicación
Postcondiciones	El usuario podrá seguir con la ejecución de la aplicación
Escenario básico	1. Se selecciona que se quiere cambiar el sufijo 2. Se introduce el nuevo sufijo

Tabla 20: Caso de Uso Cambiar sufijo

Nombre	**Seleccionar carpeta de entrada**
Actores	Usuario
Objetivo	El usuario comunica el directorio donde se encuentra el código a modificar
Precondiciones	Debe existir un código a modificar. La dirección introducida debe existir.
Postcondiciones	El usuario deberá seguir con la ejecución de la aplicación
Escenario básico	1. Se marca cambiar o no cambiar el sufijo 2. Se selecciona el directorio de entrada

Tabla 21: Caso de Uso Seleccionar carpeta de entrada

Nombre	**Seleccionar carpeta de salida**
Actores	Usuario
Objetivo	El usuario comunica el directorio donde quiere que se guarde el nuevo código que se obtenga
Precondiciones	El directorio seleccionado previamente como directorio de entrada era correcto
Postcondiciones	El usuario deberá seguir con la ejecución de la aplicación
Escenario básico	1. Se marca cambiar o no cambiar el sufijo 2. Se selecciona el directorio de entrada 3. Se selecciona el directorio de salida

Tabla 22: Caso de uso Seleccionar carpeta de salida

Nombre	**Ejecutar conversor**
Actores	Usuario
Objetivo	Conversión del código
Precondiciones	Haber superado los puntos anteriores
Postcondiciones	Ninguna
Escenario básico	1. Se marca cambiar o no cambiar el sufijo 2. Se selecciona el directorio de entrada 3. Se selecciona el directorio de salida 4. Se procede a la ejecución del conversor

Tabla 23: Caso de uso Ejecutar conversor

5.2 Diagrama de Clases UML

A continuación se muestra el diagrama de clases que se realizó en la fase de diseño, previo a la implementación del sistema.

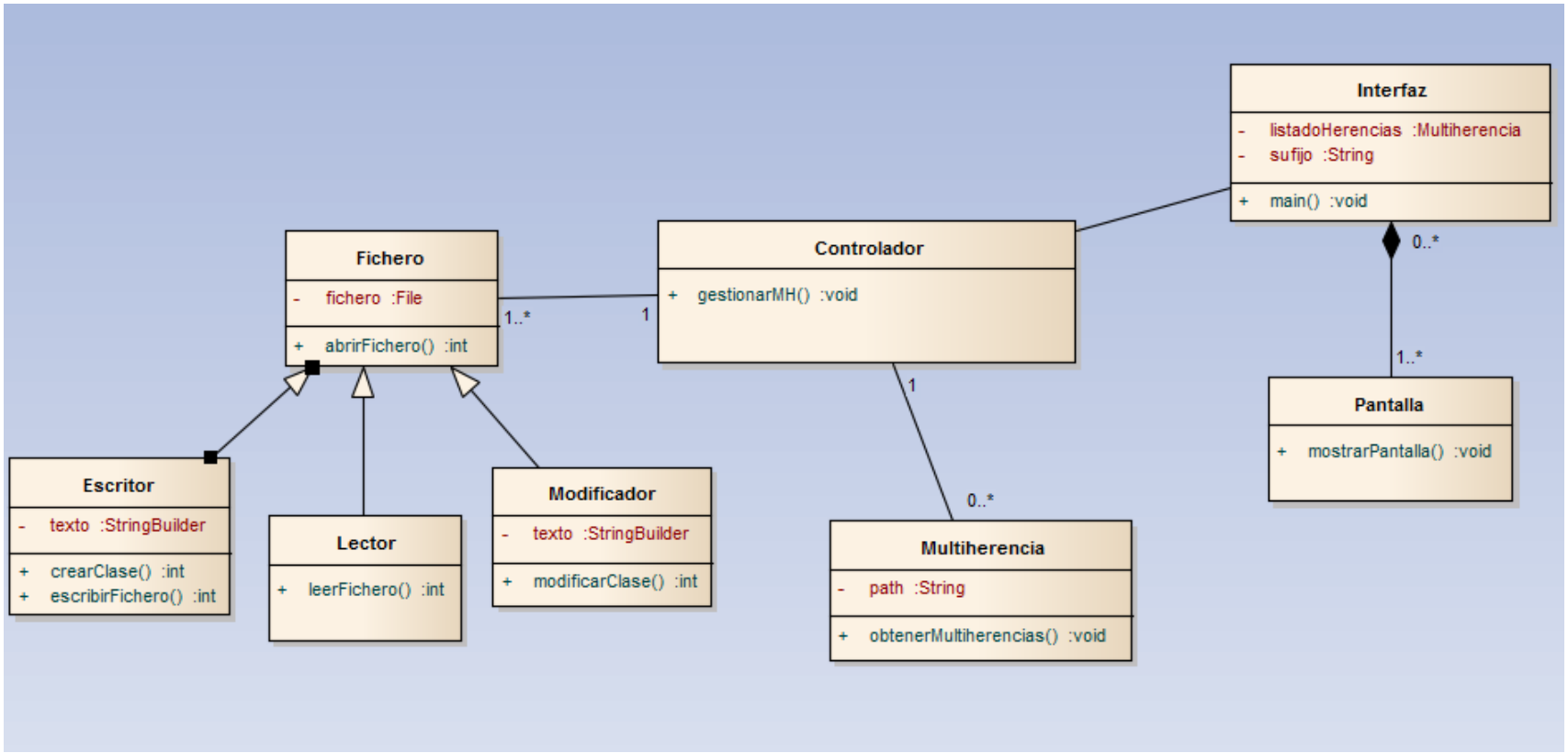

Ilustración 14: Diagrama de clases

Como se puede apreciar en la ilustración, la aplicación estará compuesta por ocho clases, que se pueden descomponer a su vez en cuatro grandes grupos:

- **Controlador:** compuesto por una única clase del mismo nombre. Es la parte encargada de unir y sincronizar al resto de clases.

- **Interfaz de usuario:** compuesta por dos clases, la clase Interfaz y la clase Pantalla. Son las encargadas de mostrar las distintas pantallas que componen la interfaz de usuario. Existe una relación de composición entre Interfaz y Pantalla, debido a que la interfaz está compuesta realmente de varias pantallas.

- **Estructural:** compuesto por una única clase, Multiherencia. Es la clase encargada de almacenar las distintas herencias múltiples encontradas, así como sus relaciones (padres, hijos, direcciones de las clases, etc.)

- **Fichero:** parte encargada de la interacción con los archivos que componen las siguientes clases, desde la copia de fichero, a su lectura, escritura de clases nuevas o a la modificación de las mismas.

Este grupo está formado por cuatro clases diferentes, la clase padre que es Fichero, con los atributos comunes como pueden ser el path y el nombre de las clases, y tres clases hijas, que se instanciarán en función del tipo de operación que vayamos a realizar. Así tenemos las tres siguientes clases hijas: Lector, Escritor y Modificador.

Como se puede apreciar, no todos los atributos y métodos que componen las clases están representados en el esquema. Para mayor sencillez, fundamentalmente de compresión del diagrama de clases, se ha representado a más alto nivel, añadiendo al esquema únicamente aquellos campos y métodos que se han considerado más relevantes o que son más útiles para ayudar a entender en términos generales el diagrama.

5.3 Arquitectura del sistema

A continuación se procederá a la descripción de la arquitectura del sistema. En primer lugar se dará una visión general del sistema para pasar a continuación a detallar cada uno de los componentes que lo forman.

A su vez cada componente puede estar compuesto por más componentes, y éstos, estarán compuestos por una o varias clases.

Antes de pasar a una explicación más concreta con los componentes de la aplicación, se comentará brevemente la composición de la aplicación. Se trata de un sistema de cuatro capas, que es una variación del sistema Modelo-Vista-Controlador (MVC), con ciertos cambios, sobre todo un menor uso de la parte del modelo y un nuevo componente, infraestructura, encargado de la interacción con los ficheros.

Este modelo es quizás excesivo para tal y como está realizada la aplicación actualmente, pero es muy efectivo de cara a futuras ampliaciones de la aplicación, ya que admitiría muy bien esas posibles ampliaciones. La descomposición que sigue el sistema se muestra a continuación:

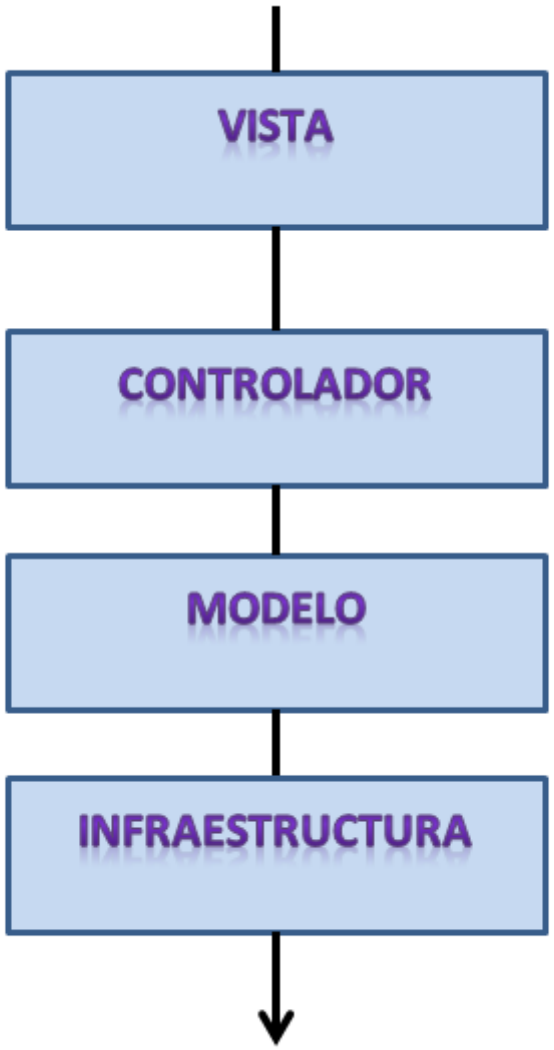

Ilustración 15: Esquema de la arquitectura del sistema

El sistema se ha dividido en las siguientes cuatro capas:

- **Vista:** Es la capa encargada de interactuar con los usuarios. Se trata de la interfaz gráfica de usuario.

- **Controlador:** Es la parte encargada de recibir los datos del controlador de la vista y comunicarse con el modelo. Es además el encargado de comprobar los datos que vienen en la vista.

- **Modelo:** Contiene el modelo de datos de la aplicación. En este caso es usado distinto a lo habitual, ya que generalmente contiene un modelo de base de datos, y en esta aplicación contiene el modelo de herencia múltiple de cada clase.

- **Infraestructura:** Consiste en todo el acceso a elementos externos a la aplicación. En este caso se trata del acceso a ficheros, pero ciertamente puede ser ampliable a otros módulos que puedan ser utilizados en un futuro si se ampliase (ver sección 9.3, Trabajos futuros).

Una vez explicado la arquitectura de manera general, procedemos a mostrar el diagrama de componentes para una visión más detallada.

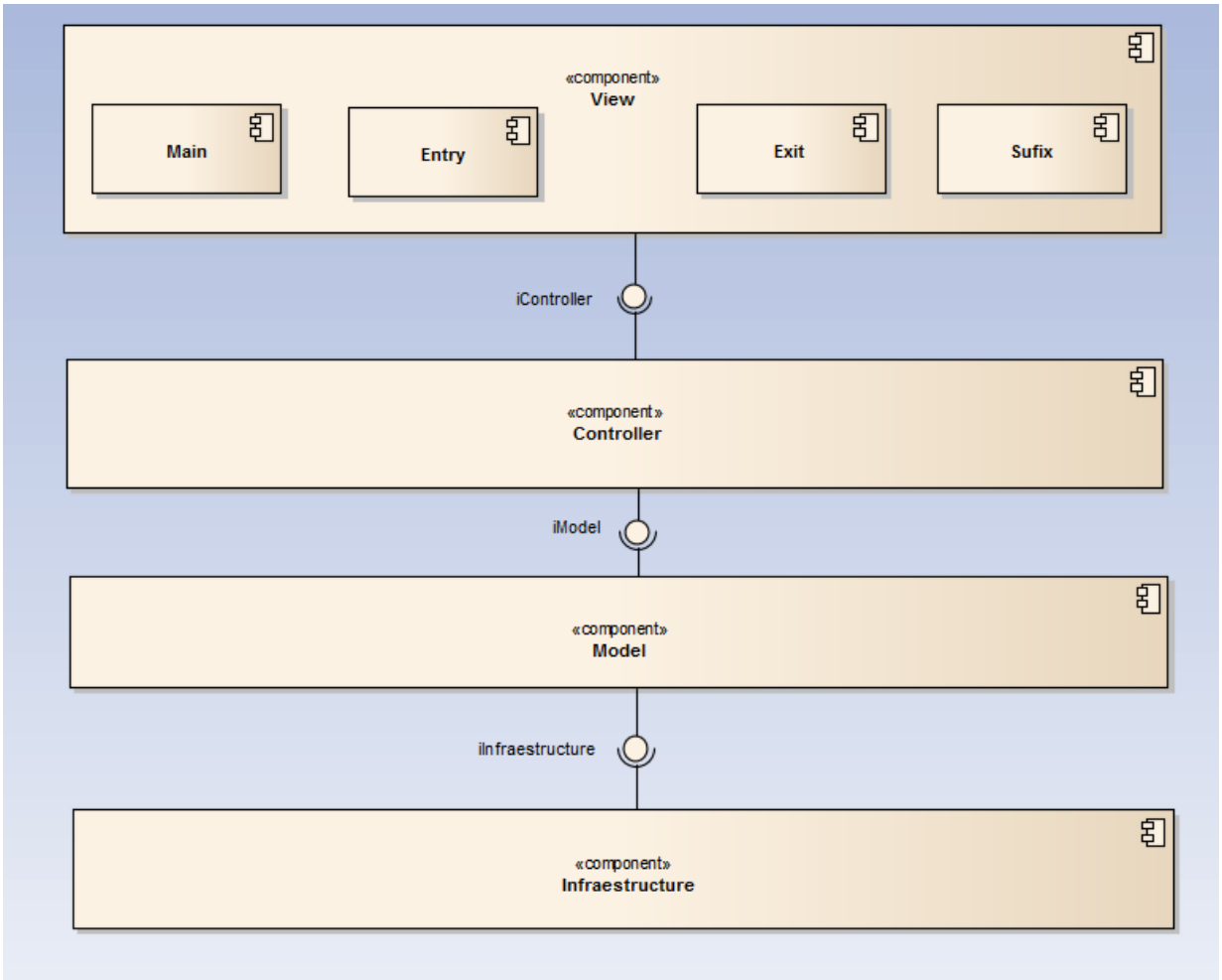

Ilustración 16: Diagrama de componentes

Viendo el diagrama anterior se aprecia como hay un componente por cada uno de las capas explicadas previamente. Igualmente se puede apreciar que algunos componentes están subdivididos además en varios componentes.

La función de cada componente es similar a la que tienen las capas de la arquitectura comentadas en este mismo apartado.

6. Detalles de la implementación

Debido a la gran importancia que tiene el código fuente en este proyecto, se ha decidido crear un apartado para explicarlo de manera general. Además, se trata de un programa muy algorítmico, y creemos que una explicación para profundizar en los distintos puntos puede ser muy interesante.

Dicho esto, la explicación será general, en este punto no entraremos a ver qué método corresponde a qué clase, no se busca algo tan detallado, si no lo que se quiere es que se entienda la algoritmia desarrollada en el proyecto y el por qué se utilizó, más que su implementación en sí.

Adicionalmente se han ido explicando alguno de los problemas encontrados durante el desarrollo de la aplicación.

Toda la aplicación, como ha sido descrito anteriormente, ha sido desarrollada en Java.

6.1 Preparación

Lo primero que nos tuvimos que plantear fue con qué interactuaría el usuario: la interfaz gráfica. Se decidió que esta interfaz fuese lo más simple posible, por comandos.

¿Por qué se optó por esta forma? El hecho de no haber realizado una interfaz más "bonita" o visual no estriba en su dificultad, ya que hacer algo de ese estilo no acarrea muchos problemas. La gracia está en que este programa pretende ser en el futuro no un programa aislado que la gente utiliza por separado, sino lo que será es parte de una serie de programas que permitirá que todo se haga de manera más fluida.

De esta manera, como este programa es una especie de precompilador, no sería descabellado su uso de manera automática cada vez que queramos compilar. Existen diversas herramientas (Ant, Maven, etc.) que permiten realizar este tipo de uso. Por tanto, las entradas del programa deberán ser lo más simples posibles, y una interfaz gráfica elaborada no solo es innecesaria, si no que puedo ser contraproducente en multitud de casos.

Otro problema encontrado fue el hecho de que para pasar de herencia múltiple a herencia simple las clases java sobre las cuales se hiciesen la conversión deberán ser modificadas, y por tanto se perderían las iniciales.

Se decidió, debido a este inconveniente, hacer una copia del original, de manera que una vez aplicada nuestra aplicación sobre el código fuente tengamos las dos versiones, la original y la nueva versión con únicamente herencia simple.

Esta copia se decidió que fuese total, esto es, no se copiarían sólo los ficheros modificados si no que se copiaría la totalidad de la aplicación a precompilar.

Una vez hecha la copia, nuestro programa pasará a modificar dicha copia, dejando intacto el código en el emplazamiento inicial.

Para realizar la copia se incluyeron dos métodos en la clase Interfaz, que son copiarFicheros y copiarDirectorios, que apoyándose en el método escribirClase de nuestra clase Escritor pasa a hacer una copia total.

Por tanto, la interfaz tendrá únicamente dos parámetros de entrada, la ruta donde está el programa que queremos precompilar y la ruta en donde queremos hacer la copia del anterior, y allí trabajar con él.

Por supuesto, como nuestra aplicación es robusta, comprobará que existe un fichero allí donde se le dice que hay y que se puede copiar a donde se le dice. En caso de no ocurrir esto la aplicación manejará correctamente el fallo, avisándonos del mismo de manera que podamos introducir de nuevo la dirección, por si anteriormente nos equivocamos.

6.2 Búsqueda de recursividad múltiple

Una vez pasado el punto anterior nuestro sistema se ejecutará totalmente de manera automática, no necesitará de nuevo la intervención de ningún usuario.

Lo que hará será buscar si tenemos herencia múltiple y en caso positivo, en qué clases.

Si no hay herencia múltiple, el programa habrá acabado su ejecución una vez confirme dicho punto.

Si encuentra herencia múltiple el programa no parará ahí, si no que seguirá recorriendo la aplicación buscando si hay más de una. Así, se podrán realizar tantas herencias múltiples como nosotros queramos, mientras que desarrollamos. La herencia múltiple estará, por tanto, totalmente aceptada en nuestro código únicamente ejecutando esta aplicación.

Una vez obtenido el listado de clases que implementan la herencia múltiple pasará a modificar la herencia múltiple para pasarla a simple, tal y como explicamos en el punto siguiente.

El algoritmo para realizar la búsqueda de la herencia múltiple es un algoritmo recursivo, muy similar al usado por los sistemas operativos para calcular el tamaño de las carpetas.

¿Qué es un algoritmo recursivo? Es aquel método, que se llama a él mismo modificando los atributos hasta que llega a un punto donde salta la condición de parada.

De esta manera, iremos entrando en cada directorio, y una vez dentro accederemos a todos sus ficheros, y en caso de haber un subdirectorio, accederemos a él de la misma manera que habíamos entrado en el anterior.

Así, finalmente, habremos accedido a todos los ficheros que hay en la carpeta que recibimos como entrada y en sus subcarpetas.

El método que hace esto, llamado leerArbol, lo podemos encontrar en nuestra aplicación, en la clase Lectura.

Por cada fichero que vayamos encontrando, veremos si es .java, y en caso de serlo pasaremos a leerlo y veremos si existe multiherencia. Esto es muy sencillo de determinar, ya que si en la línea de definición de la clase vemos que la clase hace un extends de más de una clase, es indicio inequívoco de que tenemos multiherencia.

Todas las clases que encontremos que contienen multiherencia serán guardadas en una lista, que posteriormente utilizaremos.

Además de la clase guardaremos en el mismo nodo otra información que consideramos necesaria de cara al futuro, como cuáles son sus padres.

6.3 Algoritmo para pasar de herencia simple a multiherencia

Una vez superado los dos puntos anteriores tendremos:

- Una copia de la entrada inicial.
- Una lista con todas las clases utilizan la multiherencia.

Entonces, comenzará nuestro algoritmo para pasar de herencia múltiple a herencia simple.

Este punto de nuevo será totalmente automático, como el anterior. Esto quiere decir que nuestro sistema no necesita la ayuda de ningún usuario externo.

Nótese, que en caso de no haber encontrado en el punto anterior multiherencia en ninguna clase, la ejecución del programa ya habría acabado en este punto.

También se puede observar que podemos tener varias clases con multiherencia. El sistema las tratará todas, pero de una en una. Esto es, pasará de herencia múltiple a simple la primera, y cuando haya acabado procederá con la segunda.

La forma de hacerlo viene definida en la siguiente imagen:

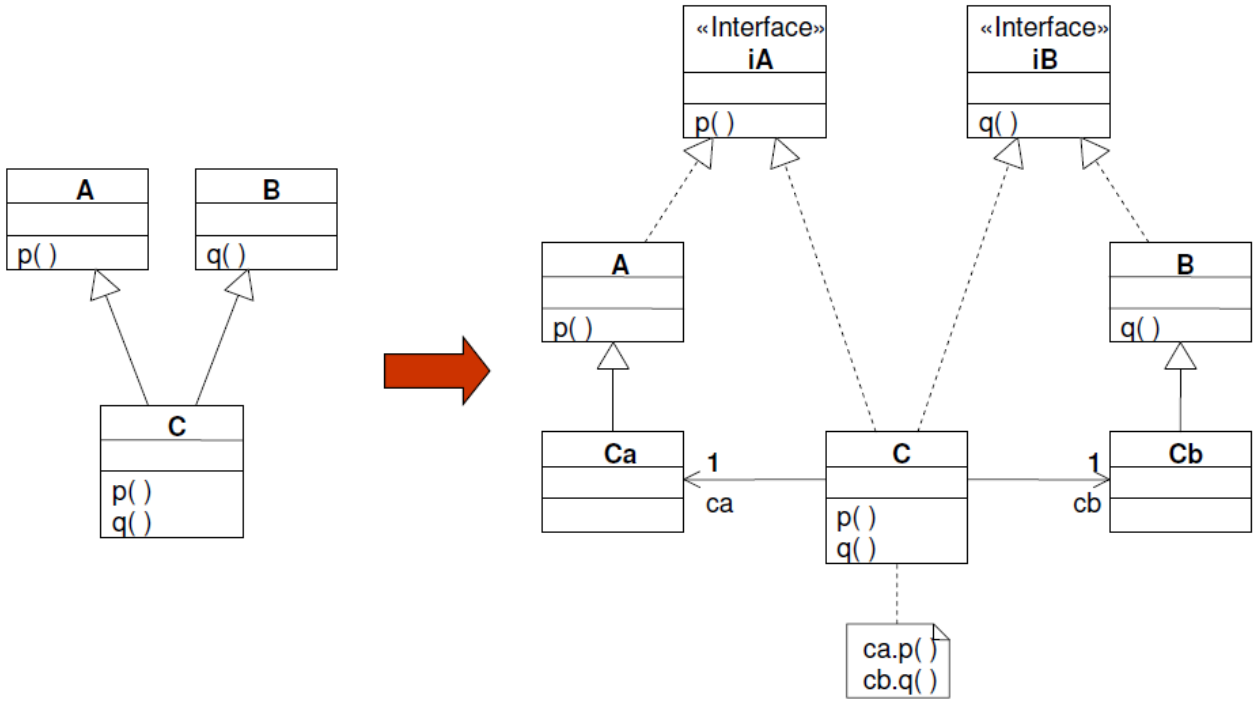

Ilustración 17: Conversión herencia múltiple [19]

Vemos en la imagen, como de una clase “C”, que implementa la herencia múltiple, se obtiene una serie de clases e interfaces que utilizan la herencia simple.

Una vez concluido este algoritmo por la aplicación que se quiere revisar, todos los errores de compilación que había en la misma producto de la herencia múltiple habrán desaparecido. De esta manera podemos pasar a ejecutar una aplicación que antes no podíamos, manteniendo toda la información semántica que proporciona la multiherencia.

El algoritmo es el siguiente:

- **Se recorre la lista que contiene las clases que implementan la multiherencia.**
  En este punto obtenemos la clase sobre la que vamos a trabajar. Sería, en nuestro gráfico, la clase “C” la que tendríamos.

  En caso de que la lista esté vacía (o ya hayamos procesado previamente todos los elementos de la lista), la ejecución se acabaría en este punto.

- **Se modifican los padres.**
  Como nos creamos una clase llamada Herencia, que está contenida en el paquete de Estructuras, cuando hemos obtenido en el paso anterior la clase que utilizaba la multiherencia, no solo obtuvimos la clase, sino que también tenemos acceso directo a más información que almacenamos previamente, como sus padres.

  Nótese en este caso, que aunque el ejemplo gráfico que hemos añadido es una herencia múltiple con dos padres, nuestra aplicación no sólo acepta ese caso, sino que además admite cualquier número de padres.

  La modificación que se debe realizar a todos los padres es la siguiente: en la línea de definición de la clase se añadirá un “implements iPadre”, que hará referencia a la interfaz que tendrá que implementar.

  En caso que la clase padre ya implementase una o varias interfaces previamente, se añadirá la interfaz “iPadre” como una más de las que tiene que implementar.

- **Se crean las interfaces**
  A continuación se pasa a crear las interfaces. Se creará una por cada padre hubiese.
  El nombre de la interfaz es el mismo del padre, añadiéndole una “i” delante, para indicar que es una interfaz.

  Además, se le ha añadido un sufijo. Este sufijo se ha utilizado para evitar que pudiese haber una clase en la misma carpeta con el mismo nombre, lo

cual sería un problema grande. Al añadir este sufijo solventamos dicho problema.

Además, este sufijo es parametrizable. Por defecto tenemos "_mh", que indica multiherencia.
Por tanto, para una clase padre llamada, por ejemplo, Vehículo, la interfaz se llamaría, iVehículo_mh.

Las interfaces deberán contener la definición de los métodos que tiene el padre, que son al final aquellos que heredó el hijo.

Para ello, al recorrer anteriormente el padre, se guardaron los métodos que éste contenía de manera que ahora se han podido añadir a la interfaz, con el formato correcto, claro.

De esta manera, una vez pasado a herencia simple, obligamos al hijo a tener todos los métodos del padre, ya que la interfaz tendrá todos los métodos del padre, y como el hijo también la implementa, obliga al hijo a tener todos los métodos, obligando a que la herencia sea "completa".

- **Se crean las clases intermedias.**
  Se han llamado clases intermedias a aquellas representadas en el gráfico como Ca o Cb.

  Al igual que en el punto anterior, con las interfaces, se hará una clase intermedia por cada padre que tenga producto de la herencia múltiple.

  Además, al igual que las interfaces, tendrán un sufijo, por defecto _mh.

  El nombre de la clase intermedia será:

  Nombre del hijo + Nombre del padre + sufijo

  Por tanto, si el hijo es, por ejemplo, Coche, y el padre es Vehículo, si tomamos el sufijo como por defecto, el nombre de la clase intermedia será CocheVehículo_mh.

  Esta clase intermedia deberá extender de la clase padre, pasará a ser ahora la clase hija. Tendrá, por tanto, por herencia, todos los atributos y métodos que tenía su padre.

  Además debe tener los constructores que sean precisos, que son aquellos que la clase padre tiene, llamando con super al constructor del padre.

  De esta manera, la clase hija inicial ("C"), podrá llamar a los métodos de esta clase (que serán los mismos del padre), de manera que será algo muy similar a lo que era antes teniendo herencia directa con el padre. Para eso fue necesario crear los constructores, de manera que la clase "C" se pueda crear un objeto de la clase intermedia y llamar a sus métodos.

- **Modificación clase hija**
  La clase hija, la llamada "C" en nuestro gráfico debe ser ampliamente modificada, también.

  Se tienen que hacer varios cambios:
  - Eliminación herencia múltiple
  - Añadir interfaces.
  - Añadir métodos "heredados".

  Eliminación herencia múltiple.
  Se eliminará de la línea de definición de la clase la herencia que haya.

  Añadir interfaces.
  En su lugar se añadirá una interfaz por cada padre que hubiese anteriormente. Estas interfaces han sido creadas, nuevas, anteriormente.

  Añadir métodos heredados.
  Se deben añadir todos los métodos que tiene la clase padre. Así mismo son los métodos que tiene la interfaz que implementa la clase hija, luego es necesaria su implementación.

  Para ello lo que se hace es, teniendo un listado de los métodos que son, implementar dichos métodos llamando al método que realmente tiene la funcionalidad del mismo. En herencia ese método sería el del padre, en este caso es el método de la clase intermedia (que por herencia tiene el del padre).

  Para tal fin crearemos un objeto del tipo de la clase intermedia. Esto es posible ya que en esta clase intermedia nos creamos los constructores (públicos) necesarios. Al llamar a la clase intermedia, y como ésta hereda de la clase padre, tendremos la funcionalidad.

  De esta manera tenemos una funcionalidad equivalente a la herencia múltiple, pero únicamente usando la herencia simple e interfaces, tal y como hemos comentado antes. Y la relación, aunque un poco más compleja, mantiene el significado semántico que daba la herencia múltiple.

  Como hemos dicho, finalmente la clase hija acabará "heredando" todas las funcionalidades que sus padres tienen, tal y como haría si heredasen de ellos.

6.4 Dificultades comunes a todo el proceso

Existen diversas dificultades comunes a todo el proceso, que han sido solventadas a través de la programación.

Destacan, entre los problemas, los siguientes:

- Comentarios
- Una línea escrita en varias
- Paquetes
- Modificadores de acceso

Comentarios

Cuando estás buscando en las clases java distintos comandos, hay que tener cuidado cuando lo encuentres que lo que parece un comando válido no sea en realidad un comentario.

Por ejemplo, una clase podría tener herencia múltiple como un comentario, por tanto no lo tendría realmente. El programa debe distinguir esos casos.

Para ello tenemos un método esComentario que devuelve un booleano en la clase Lector.

Ejemplo de comentario:

```
/*
* Public class Fresador extends Trabajador, Persona
*/
```

Una línea escrita en varias

Java tiene la propiedad de poder introducir tantos intros como se quieran, sin modificar el significado de la línea.

Esto es, una línea podría estar realmente escrita en n líneas, siendo n cualquier número.

Además, en adición al punto anterior puede haber comentarios intercalados entre las distintas líneas vacías. El programa discriminará esos comentarios y los intros introducidos.

Paquetes

En ocasiones las clases padre e hija pueden estar en diferentes paquetes. Esto puede dar lugar a problemas de visibilidad. Hay que tener en cuenta esto cuando se crean nuevas clases.

Modificadores de acceso

En adición al punto anterior, clases en diferentes paquetes, tenemos los modificadores de acceso.

Los atributos o métodos que sean privados, no se heredarán. Igualmente, los que no tengan modificador de acceso, default, tampoco se heredarán si están en diferentes paquetes.

La aplicación contempla estas eventualidades.

6.5 Clases implementadas

A continuación se pasa a describir de manera ligera el diagrama de clases de la aplicación.

Se puede observar que se trata de un diagrama muy ligero. Esto es debido a que no tiene clases de "relleno", esto es, que como no tiene acceso a bases de datos no existen clases para este fin. Hay ocasiones que este tipo de acceso a base de datos originan muchas clases, debido a que a veces se implemente una clase por tabla incluso.

Además, tampoco se ha implementado una interfaz compleja, dando lugar a necesidad a un conjunto importante de clases para este aspecto.

De hecho, la mayoría de clases y métodos que constan en la aplicación son para algoritmia y lógica, que es la parte central de la aplicación. Se ha intentado ser conciso, y usando la recursividad se ha conseguido un código bastante elegante y compacto, necesitando para hacer la totalidad de la funcionalidad deseada un código bastante reducido, sin mucho volumen de clases.

A continuación se muestra un diagrama de paquetes que muestra la aplicación de manera general.

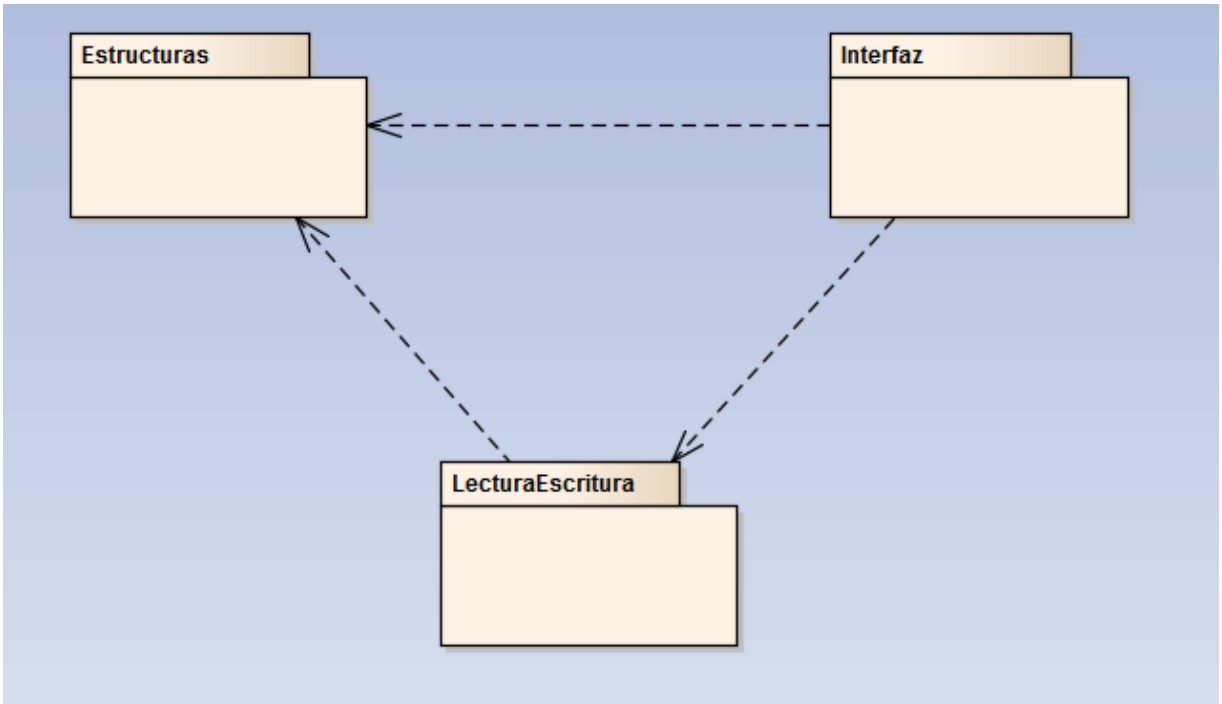

Ilustración 18: Diagrama de paquetes

Se puede observar en la imagen superior cómo la aplicación está compuesta por tres paquetes: *Interfaz*, *Estructuras* y *LecturaEscritura.*

La relación entre los distintos paquetes es la siguiente:

- Desde el paquete Interfaz se puede acceder a los demás paquetes. Esto es lógico ya que Interfaz es un poco el paquete desde el cual surge todo.

- Además, también se puede consultar el paquete *Estructuras* desde el paquete *LecturaEscritura*.

A continuación se irá entrando en cada paquete por separado, explicando un poco más la funcionalidad de cada uno de ellos.

6.5.1 Paquete Interfaz

El paquete Interfaz es el encargado de, como se puede adivinar por su nombre, interactuar con el usuario.

Además, también realiza las funciones propias del método main que además posee, esto es, hace un poco de centro neurálgico de la aplicación, distribuyendo el trabajo a los distintos métodos de ésta y otras clases, siendo un poco el "organizador" de la aplicación.

El paquete Interfaz está compuesto por una única clase, también llamada Interfaz.

A continuación se muestra una imagen con los atributos y métodos de la clase de este paquete.

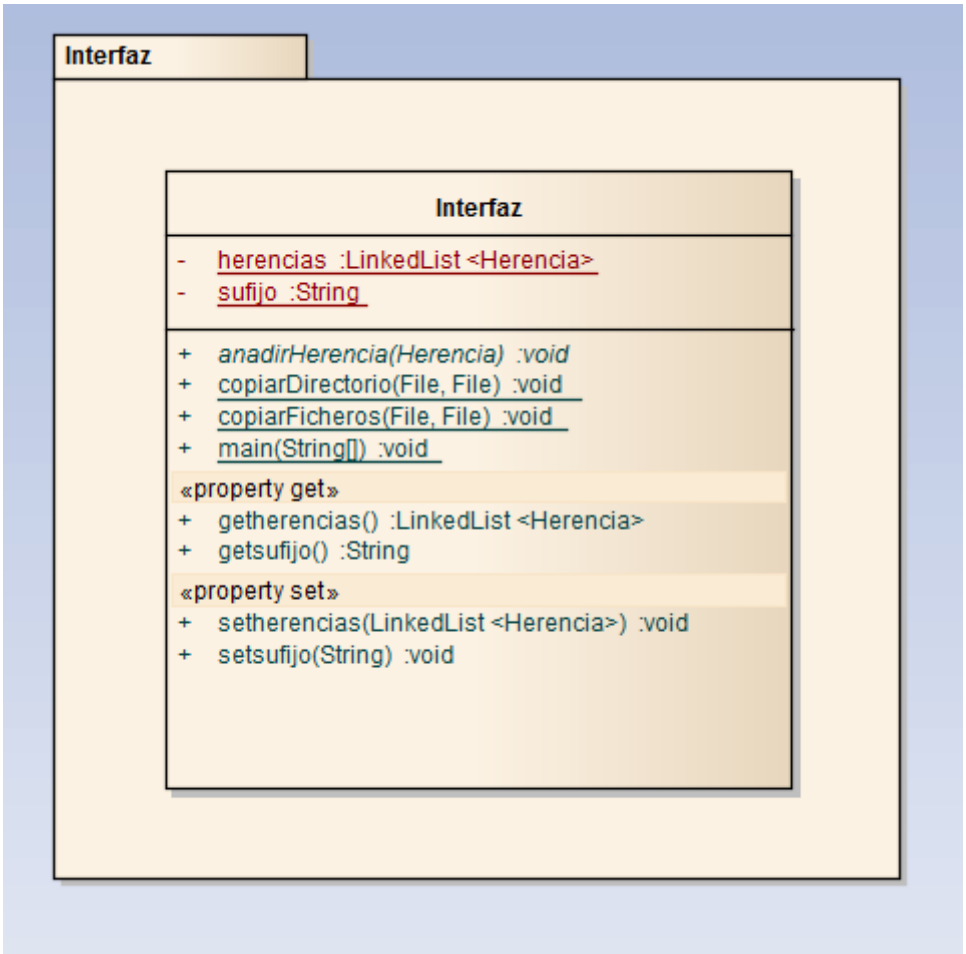

Ilustración 19: Paquete Interfaz

Los puntos más destacados de esta clase son:

- El listado de herencias. Se trata de un atributo donde se guardan para cada clase que tiene herencia múltiple todas las relaciones que tiene con otras clases, como pueden ser todos sus padres, por ejemplo. Esta lista será consultada después desde otros paquetes.

  Se trata de una referencia a una clase creada en el paquete *Estructura*.

  Para crear ese listado se dispone del método *anadirHerencia*.

- El atributo sufijo. Será el sufijo que se añada a las clases de nueva creación. El sufijo por defecto es "_mh".

- El método principal. El main será el distribuidor de trabajo respecto a toda la aplicación.

  Además, contiene toda la interfaz del usuario.

- Los métodos de copia de la aplicación. Cuando se procede a analizar una aplicación Java se realizará una copia para no modificar el original y hacer todos los cambios sobre una nueva aplicación, que será una copia exacta de la original.

  Los métodos que hacen esta copia son:

  - copiarDirectorio: para las copias de carpetas.
  - copiarFicheros: para la copia de archivos.

6.5.2 Paquete Estructuras

El paquete Estructuras es el encargado de dar formato a las herencias múltiples de forma que la aplicación pueda, en el futuro, entender dichas herencias y procesarlas según convenga.

Este paquete contiene una única clase, la clase Herencia, que contiene los datos que se deben tener de cada herencia múltiple.

A continuación se muestra un diagrama con los datos de dicha clase.

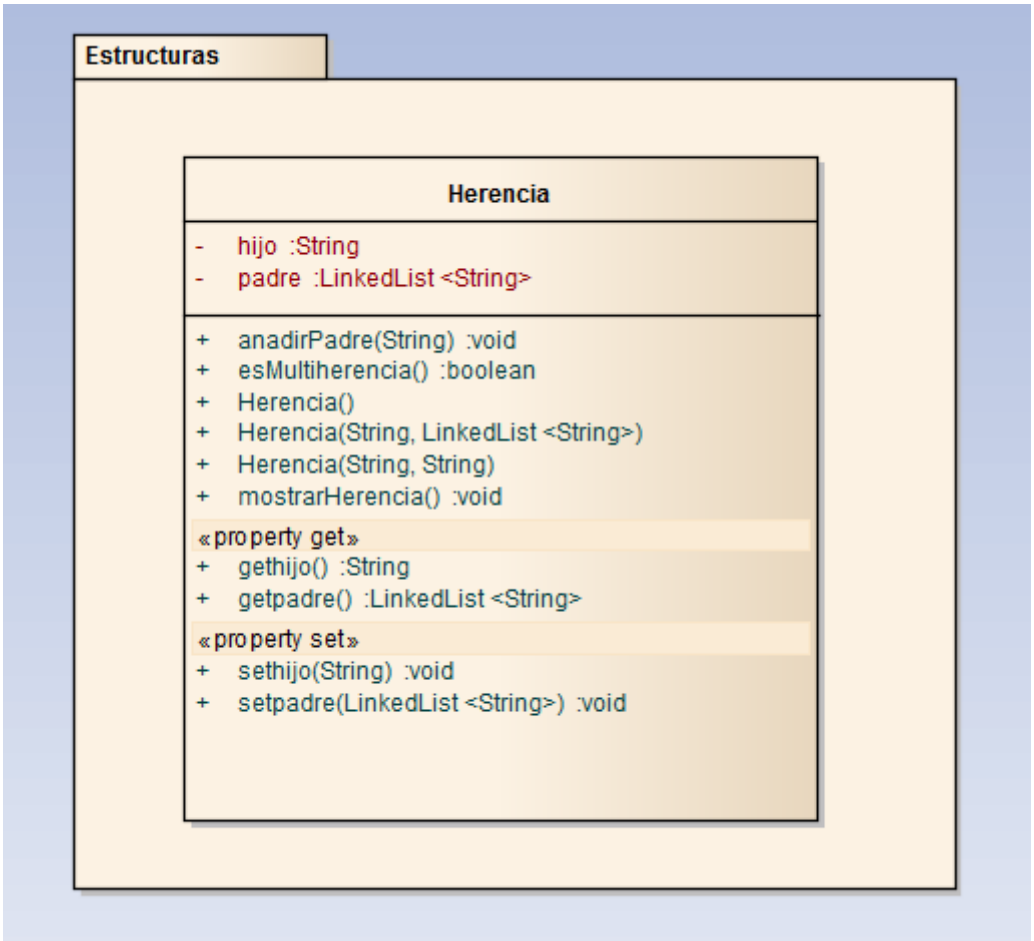

Ilustración 20: Paquete Estructuras

Los puntos más destacados de esta clase son:

- El atributo hijo. Por cada herencia múltiple se guardará quien es el hijo de la relación.

- El listado de padres. Se trata de un atributo donde se guarda un listado de todos los padres que hay en cada herencia múltiple,

guardando su relación con los demás padres y con su hijo, por supuesto.

- Además, de manera dinámica se pueden ir añadiendo más padres a la relación, con el método *anadirPadre*.

6.5.3 Paquete LectorEscritor

El paquete LectorEscritor es el encargado de interactuar con las distintas clases, en aspectos tales como leerles, crear clases nuevas, eliminar datos de algunas de ellas y añadir información en otras.

Este paquete está compuesto por tres clases Java: *Lector*, *Escritor* y *Modificador*.

❖ Clase Lector

La clase Lector tiene como función principal el acceso y lectura de las clases de la aplicación.

A continuación se muestra la clase.

Ilustración 21: Paquete LectorEscritor, clase Lector

Los puntos más destacados de la clase Lector son:

- Referencia a la clase Herencia. De nuevo se tiene una referencia aquí.

- El atributo raíz. Nos da la dirección url del fichero que estamos leyendo.

- El método leerArbol. Se trata de un método fundamental dentro de la aplicación. Es el método encargado de rellenar el listado de herencias por cada clase hija.

Este método es llamado por el método main de la clase Interfaz, una vez que la aplicación a analizar ha sido copiada.

Este método, leerArbol, es el que recorre recursivamente toda la aplicación. El método encargado de leer cada clase y buscar sus herencias es leerClase.

- El método esComentario. Es importante saber cuándo se está analizando una clase si la línea que se está analizando es real o se trata de un comentario. Este método discrimina eso.

❖ Clase Escritor

La clase *Escritor* tiene como funcionalidad principal la escritura en ficheros, principalmente de partes nuevas. Si es algo que ya existía se suele usar la clase Modificador.

También se utiliza la clase *Escritor* cuando se crea una nueva clase, como por ejemplo las nuevas interfaces que se hacen o las clases intermediaras entre el hijo y el padre que se crean posteriormente, que también llamamos clases "c".

A continuación se muestra la clase *Escritor*.

Ilustración 22: Paquete LectorEscritor, clase Escritor

Los puntos más destacados de la clase *Escritor* son:

- El método AniadirConstructores. Crea un nuevo constructor allí donde sea necesario ahora. Esto suele utilizarse cuando se quita la herencia, lo que antes era clase hija ahora pasa a necesitar constructor, ya que ahora no lo hereda.

- El método escribirClase. Dado un listado de Strings con la información que queremos que contenga la clase, este método crea la clase.

  Se utiliza también para crear las nuevas interfaces.

❖ **Clase Modificador**

La clase *Modificador* tiene como funcionalidad principal la escritura en fichero para algo que ya existiese previamente, modificando por tanto dicho fichero.

A continuación se muestra la clase *Modificador*.

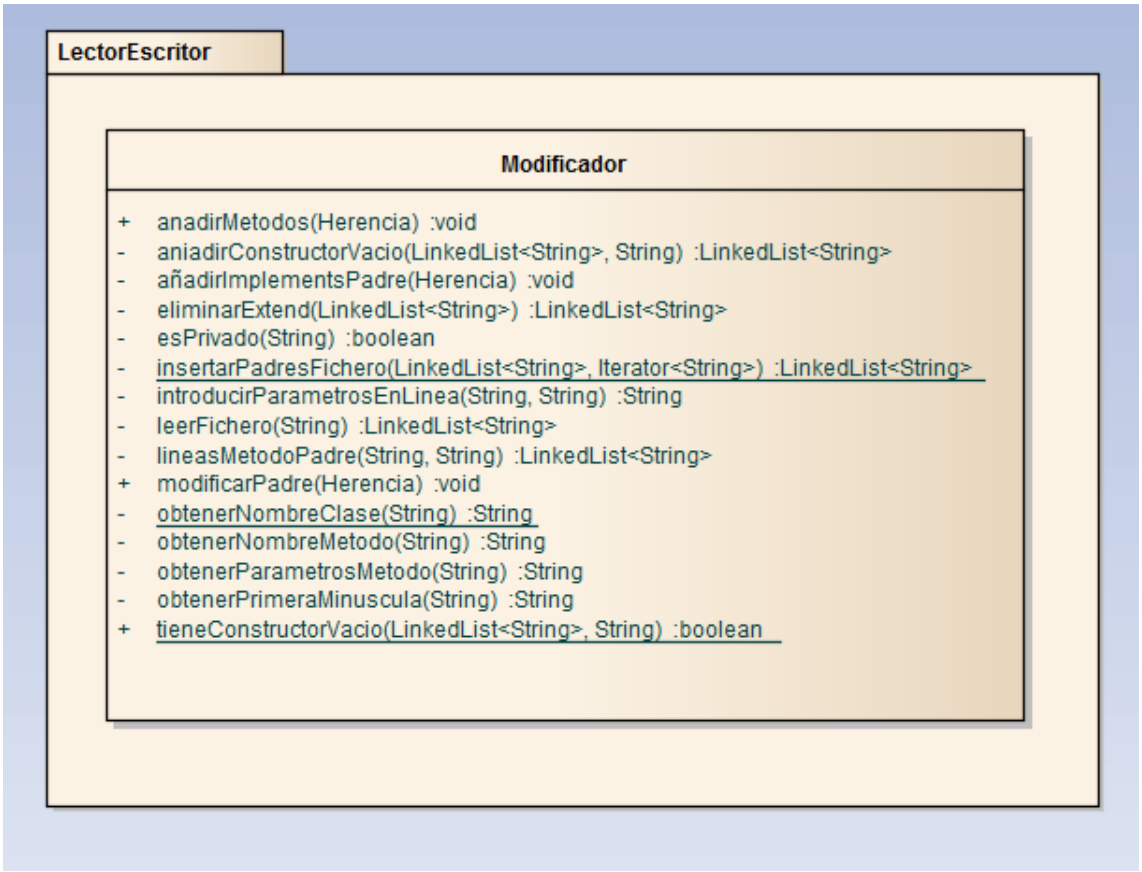

Ilustración 23: Paquete LectorEscritor, clase Modificador

Los puntos más destacados de la clase *Modificador* son:

- El método AniadirMetodos. Se añade a la clase hija la llamada a los métodos de las clases padres.

  Estas llamadas antes no eran necesarias ya que la clase hija lo heredaba, pero ahora se necesitará.

- Métodos de modificación de extends e implements. Hay varios métodos que son usados para modificar estas partes de la clase. Ejemplos: aniadirImplements, eliminarExtend…

- Métodos para modificar los métodos. Son un conjunto de métodos que permiten añadir o modificar un nuevo método en una clase.
  Entre estos métodos encontramos obtenerNombreMetodo, que devuelve el nombre y referencia del método, obtenerParametrosMetodo, que te da todos los parámetros del método que vamos a eliminar, o heredar, etc. Y el método introducirParametrosEnLinea¸que dado los parámetros que queremos añadir los incorpora en el método que queramos.

7. Pruebas del Sistema

En cualquier producto software que se desarrolle hay que realizar una exhaustiva fase de pruebas. Esta es una de las fases más importantes en cualquier desarrollo, y a su vez una de las más olvidadas en relación a su importancia.

Esta fase de pruebas engloba a su vez distintas acciones, tales como la definición y ejecución de las mismas, así como la futura validación de los resultados.

Las pruebas deben estar orientadas a ver los posibles fallos del software, de manera que, si la pasan, se pueda entregar una aplicación de calidad, que cumple con aquello que dijo que haría.

En este apartado se han detallado las distintas pruebas que se han realizado en este proyecto.

Hay que tener en cuenta que como en este caso estamos tratando de un proyecto no profesional, las pruebas no han podido ser tan amplias como sin duda lo hubiesen sido en un entorno laboral, con posiblemente más recursos, sobre todo personal, para la realización de un paquete más completo de pruebas.

Además, en el ámbito profesional lo más adecuado es que las personas que realizan las pruebas sean distintas a las personas que han desarrollado el código. Obviamente esto aquí ha sido imposible de llevar a cabo.

Aun así, en el entorno en el que nos movemos, las pruebas han sido bastante completas, siempre sabiendo que las pruebas, aún muy importantes para encontrar fallos, no te aseguran la ausencia total de éstos, ya que la prueba total es imposible.

A continuación se muestran los distintos tipos de pruebas que se han realizado en este proyecto.

7.1 Tipos de pruebas

Hemos realizado dos tipos de pruebas: pruebas estáticas y pruebas dinámicas.

Para que las pruebas sean llevadas a cabo de manera satisfactoria no es suficiente con hacer las pruebas al final, una vez tengamos finalizado el código, si no que a lo largo del desarrollo de éste se deben ir haciendo distintas pruebas que nos facilitarán las pruebas finales, ya que es mucho más fácil encontrar un error sobre una pequeña porción de código que sobre la totalidad de éste.

Además, no es el código la única parte del proyecto que se debe probar o revisar, si no que tanto el diseño como la documentación deben ser checkeados.

En estos apartados se tratarán las pruebas y se dirán cuales se han hecho sin entrar mucho en profundizar sobre ellas. Tan solo al final, con un ejemplo de una prueba "final" se verá un ejemplo de pruebas realizadas sobre el desarrollo.

7.1.1 Pruebas estáticas

Son aquellas pruebas que se realizan sin tener que ejecutar el código. También son llamadas revisiones.

Estas pruebas son realizadas en las primeras fases del desarrollo, especialmente en las fases de análisis y diseño, aunque en fases de implementación también pueden ser realizadas.

Se han acometido las siguientes pruebas:

- **Pruebas estáticas sobre el diseño.**
  Se ha comprobado el diseño inicial así como las distintas modificaciones, que se ha realizado a lo largo del desarrollo del proyecto, sobre el diseño.

  Además se ha mirado que el resultado final efectivamente cumpliese con el diseño realizado.

- **Pruebas estáticas sobre el código.**
  Estas pruebas han sido desarrolladas en tiempo de implementación. Consiste en una revisión visual del código, en muchos casos tras imprimir en papel dicho código. Hay estudios que afirman que a través de esta prueba es la forma en la que más errores en el código se pueden encontrar, en porcentaje sobre el total. Aunque normalmente se tratan de pequeños errores, en muchos casos este tipo de revisión te puede ahorrar mucho tiempo en otras pruebas futuras.

En muchos casos esta prueba ha sido combinada con las pruebas unitarias, que se explicarán más adelante.

- **Pruebas estáticas sobre los documentos.**
  Se han realizado diversas revisiones sobre los distintos documentos asociados al desarrollo, para asegurar una calidad sobre dichos documentos.

  Estas pruebas destacan por haber sido las únicas que no sólo han sido realizadas por el alumno, si no que el tutor también ha formado parte de ellas.

7.1.2 Pruebas dinámicas

Son pruebas dinámicas aquellas que para su realización exigen la ejecución del objeto que se está probando.

A continuación se explican las distintas pruebas dinámicas que se han realizado.

- **Pruebas unitarias.**

  Estas pruebas han sido realizadas a lo largo de todo el proceso de desarrollo.

  Consiste en probar cada componente de manera individual, hasta el punto de ir probando método a método. Obviamente, hay métodos que no se han probado por su simplicidad, como pueden ser los getters y los setters.

  Cada método se ha ejecutado en repetidas ocasiones modificando la entrada al mismo. Se ha comprobado que la salida del método es la correcta (pruebas de caja negra).

  Además se ha comprobado por cada método que así lo requiriese las posibles bifurcaciones que puede tomar en función de los datos de entrada (pruebas de caja blanca).

- **Pruebas de integración.**

  Estas pruebas consisten en probar los distintos componentes una vez se han unido (integrado).

  Esto es debido a que aunque sepamos que los componentes por si solos funcionan de manera correcta, esto no es un indicador fiable de que de manera conjunta dichos componentes sigan trabajando correctamente.

Para ello se procedió a hacer una batería de pruebas similar al caso anterior, dando distintas entradas a los métodos, ya pertenecientes a componentes integrados, y viendo si la salida corresponde a lo esperado.

- **Pruebas del sistema.**

  Cuando el sistema está finalmente desarrollado se procede a realizar este tipo de prueba. Se trata de probar la aplicación "como un todo". Es una extensión a la prueba de integración.

  Para ello se probará no ya que los métodos hagan lo que tienen que hacer, si no que sea la aplicación la que lo haga. Se comprueba que se haya implementado todos los casos de uso, y que estos estén realizados de manera correcta. Además deberá probarse que la aplicación sea robusta, y se recupere bien tras errores o excepciones.

  En el punto siguiente veremos un ejemplo detallado de este tipo de prueba.

- **Pruebas de implantación.**

  Estas pruebas consisten en ver cómo funciona la aplicación en distintos ordenadores, con distintos sistemas operativos, con distinta configuración, etc.

  Para ello se ha instalado la aplicación en diversos ordenadores, y se ha procedido a la realización de las pruebas del sistema en cada uno de ellos.

  No se observó ninguna diferencia en la salida a dichas pruebas en ninguna máquina.

  Estos resultados eran esperables, porque este desarrollo no depende de ninguna aplicación, depende únicamente de Java. Y como se comentó en el estado de arte, Java es independiente de la plataforma.

- **Pruebas de aceptación.**

  Este tipo de prueba consiste en que el usuario final interactúe con la aplicación, dando su visto bueno a la misma.

  Obviamente, en este tipo de aplicaciones esto no es posible, ya que el cliente final no es un cliente concreto al cual podamos tener acceso. Por tanto se ha probado con lo más parecido que se ha podido obtener, distintos programadores con conocimiento de Java que podrían ser clientes potenciales de la aplicación en el futuro cercano.

La prueba ha sido realizada tomando a estos programadores como clientes, esto es, como usuarios de la aplicación.

- **Pruebas de regresión.**

  Estas pruebas están pensadas para el mantenimiento, pero deben realizarse con anterioridad, al mismo tiempo que el resto de las pruebas.

  Las pruebas tienen como objetivo el ver cómo un cambio en un módulo en el futuro afectaría a otros componentes de la aplicación.

  Dichas pruebas han sido realizadas conjuntamente con las pruebas de integración, debido al parecido funcional existente entre ambos tipos de pruebas.

7.2 Datos de prueba

Se han realizado cuatro pruebas del sistema. Tres de ellas son de un carácter más sencillo y existe una cuarta de una mayor complejidad.

Esta última prueba se explicará en mayor profundidad en el siguiente punto.

Comentar que todas las pruebas (no sólo las de sistema) han sido realizadas usando la versión 7 de Java. En un único caso, el más complejo, se ha usado la versión 8, obteniendo un resultado satisfactorio (exactamente el mismo que se obtuvo con Java 7).

A continuación se muestra una tabla con los cuatro conjuntos de pruebas ejecutados:

Nº clases Java	Nº clases multiherencia	Nº padres
3	1	2
4	1	3
3	0	1
13	2	2/3

En todos los casos el resultado obtenido por el programa es el esperado. Como se ha comentado previamente, debido a su mayor complejidad y a su idoneidad para tal efecto, la última prueba marcada se explicará en profundidad en el siguiente punto.

7.3 Ejemplo prueba del sistema

Una vez explicadas todos los tipos de prueba que se han llevado a cabo en nuestra aplicación y habiendo profundizado ligeramente en las pruebas del sistema pasaremos a explicar de manera más específica la prueba del sistema más completa que se ha realizado.

¿Por qué se ha optado por explicar este tipo de prueba de manera detallada y no las demás?

La respuesta es sencilla. Se ha considerado que era muy interesante explicar esta prueba en mucho detalle, a diferencia de las otras por estos dos motivos:

- **Imposibilidad de explicar en detalle todas las pruebas.**

  Todas las pruebas explicadas previamente han sido llevadas a cabo, pero es del todo imposible comentar todas en detalle ya que daría lugar a un punto demasiado extenso, de manera totalmente innecesaria.

- **Prueba más interesante. Permite además explicar el proceso de pasar de herencia múltiple a herencia simple con un ejemplo.**

  Esto es lo realmente interesante de esta prueba. Es con mucha diferencia la prueba más interesante de todas, primero porque engloba a muchas otras pruebas (pruebas unitarias o de integración, por ejemplo, están totalmente incluidas en esta prueba), ya que viendo el resultado de esta prueba, si es correcto, podemos aventurar que el programa funciona correctamente. Por supuesto, para llegar al punto en el cual se ha podido realizar esta prueba ha sido gracias a que anteriormente se realizaron las otras.

  Por supuesto, aunque en este caso solo se muestre un único ejemplo de prueba del sistema, en la fase de pruebas de la aplicación ha sido probado más de un único caso, aunque mostrar más de uno se ha considerado innecesario, por reiterativo.

  Y el punto principal por el cual se muestra la prueba del sistema, el punto principal del proyecto, de hecho, es para poder explicar de nuevo el paso de la multiherencia a la herencia normal que admite Java, o herencia simple.

  Hasta ahora en este documento dicho paso ha sido ampliamente explicado, pero siempre desde un punto de vista teórico. Ahora se

podrá explicar de manera práctica, viendo la siguiente prueba, que hará las veces de ejemplo práctico.

El siguiente ejemplo parte de la premisa de ser un código Java, que simula a una aplicación real (de hecho es una aplicación realmente, no muy útil, pero totalmente operativa, de manera teórica, en Java) que hace uso de la herencia múltiple.

El programa tendrá que tomar esta entrada y precompilarla, esto es, modificarla de manera que se eliminen los errores de compilación, pero manteniendo la esencia del mismo.

Pasamos ahora a explicar el programa que se ha de modificar. Veremos un poco su estructura, y luego analizaremos sus dos paquetes principales, que, al no estar relacionados entre ellos, se podrían considerar dos ejemplos aislados.

7.3.1 Estructura general de la prueba

La prueba consiste fundamentalmente en un programa Java. Se trata de un programa que compila perfectamente, a excepción de los casos que hacen uso de la herencia múltiple, que como ya se ha comentado previamente en el documento, no está aceptada por Java.

Esta aplicación en caso de querer ejecutarse se podría hacer, aunque bien es cierto que lo interesante de la aplicación es su estructura, que es totalmente correcta. Respecto a su funcionalidad es inexistente, ya que el hecho de que hagan los métodos de prueba es algo indiferente para la aplicación que se está probando.

Respecto a los ficheros que contiene, todos son ficheros .Java. Se ha probado además su interacción con otro tipo de ficheros, y la reacción de la aplicación a ellos es la esperada, los copia. No se han incluido en este ejemplo concreto (recordemos que se han hecho varios ejemplos) ya que su aportación para el entendimiento de la aplicación es indiferente.

Además de los ficheros, el programa contiene paquetes. Un paquete, al menos en el lenguaje de programación Java, es una entidad que contiene diversas clases u otros ficheros del programa, permitiendo agrupar el código en distintos “módulos” en función de si comparten funcionalidades. A nivel físico no son más que carpetas.

A continuación mostramos el directorio raíz del programa de prueba que analizaremos.

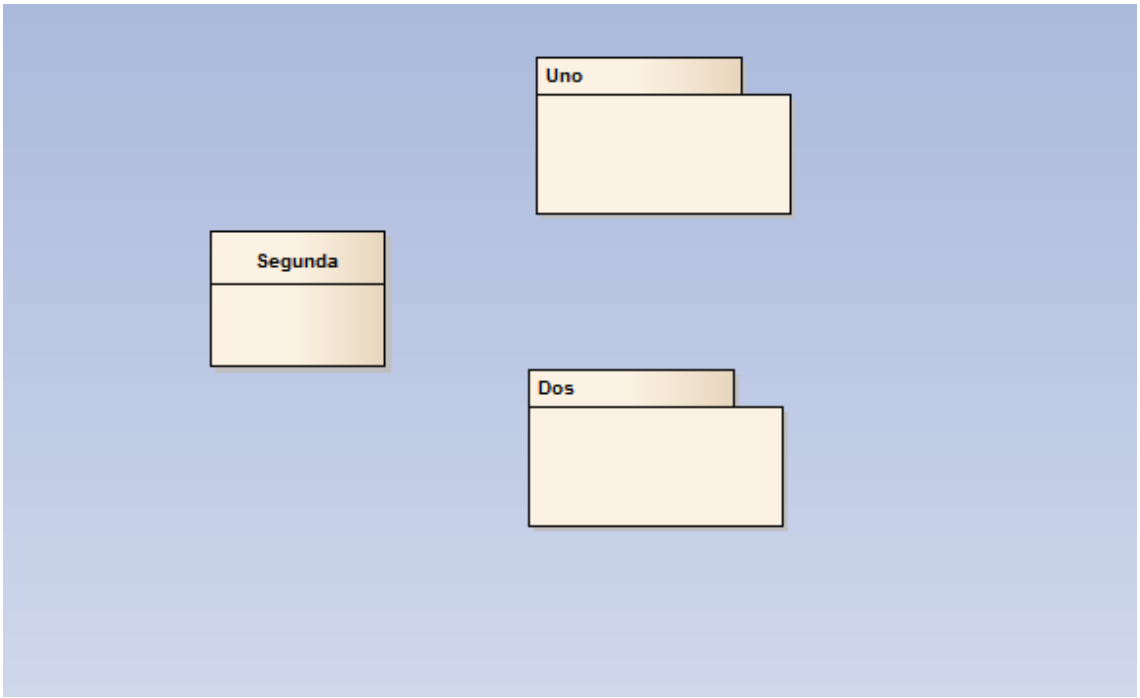

Ilustración 24: Programa de prueba

Así, a bote pronto, se puede ver que el directorio raíz está compuesto de los siguientes elementos:

- Una **clase Java:** Segunda.
- Dos **paquetes**: Uno y Dos.

Respecto a los nombres de las clases y los paquetes, como este programa a probar ha sido creado expresamente para este fin, éstos han sido puestos de manera que permitan una trazabilidad de las distintas clases con respecto a sus paquetes lo más sencilla posible.

También podemos observar que los métodos en las distintas clases no han sido especificados en el diagrama. Por supuesto las distintas clases tienen métodos, y estos han sido comprobados para ver que han sido modificados o creados de manera correcta, pero en este ejemplo han sido omitidos para hacerlo más sencillo y fácil de seguir, y centrarnos en lo que creemos realmente importante.

Lo mismo que ha sido dicho para los métodos aplica para los atributos.

Volviendo al ejemplo, la única clase, como se puede apreciar, no tiene ningún tipo de herencia múltiple (no tiene ningún tipo de herencia, de hecho), y por tanto la aplicación no debe modificarla.

Ahora se pasará a analizar los dos paquetes por separado, que es donde está el meollo de la prueba. La aplicación, como se comentó anteriormente en este documento, recorrerá todos los paquetes de manera recursiva, hasta acceder a todos los ficheros del programa de pruebas, de manera que puedan ser analizados y modificados, además de crear ficheros nuevos, en caso de ser preciso.

7.3.2 Prueba: paquete Uno

A continuación se muestra un gráfico con la composición del primer paquete a analizar:

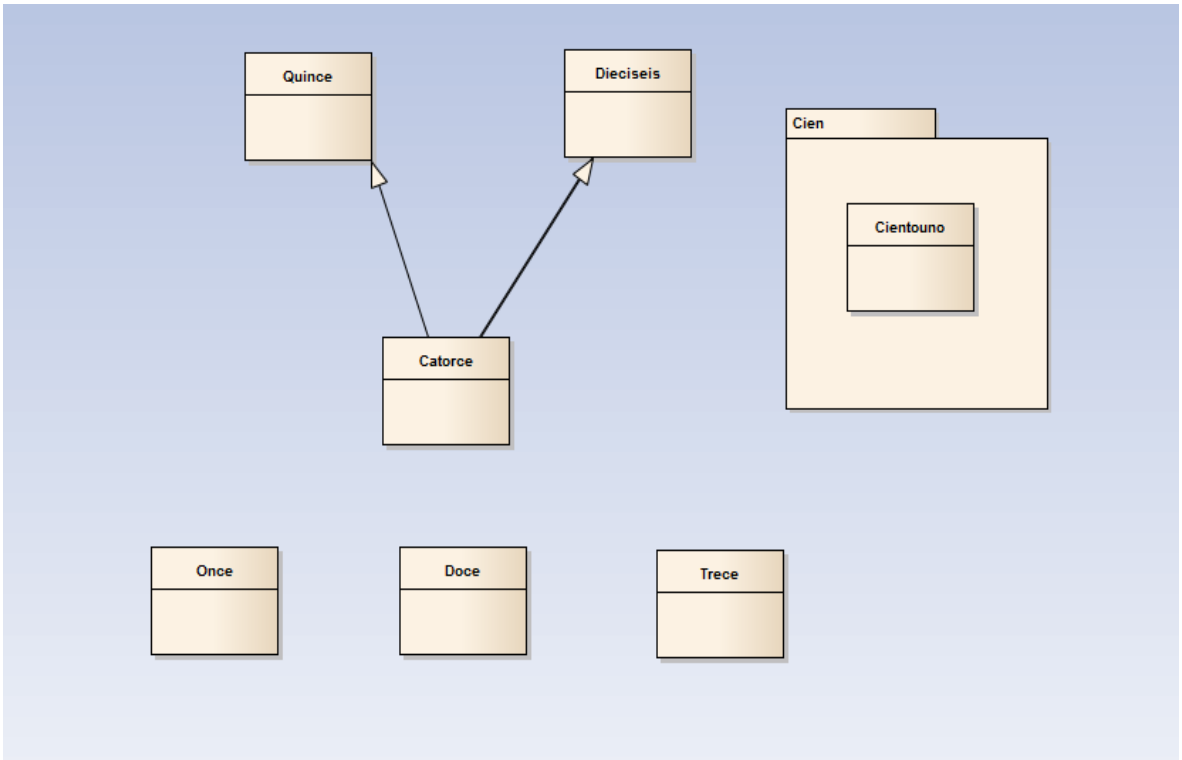

Ilustración 25: Paquete Uno

Destacan en este paquete:

- Tenemos **herencia múltiple**. Las clase *Catorce* es hija de las clases *Quince* y *Dieciséis.*

- Existen tres clases sin ningún tipo de conexión. Esas clases deberán ser copiadas sin modificaciones. Nos referimos a las clases *Once*, *Doce* y *Trece.*

- Existe un **paquete interno.** Se trata de *Cien*, que a su vez tiene una clase dentro, *Cientouno.* Este paquete será tratado en este mismo punto.

Como se comentó en el estado del arte, la conversión que se realizará de la herencia múltiple a herencia simple es una modificación de las clases implicadas, añadiendo además otras clases nuevas e interfaces.

A continuación recordamos con la siguiente imagen la conversión que la aplicación lleva a cabo.

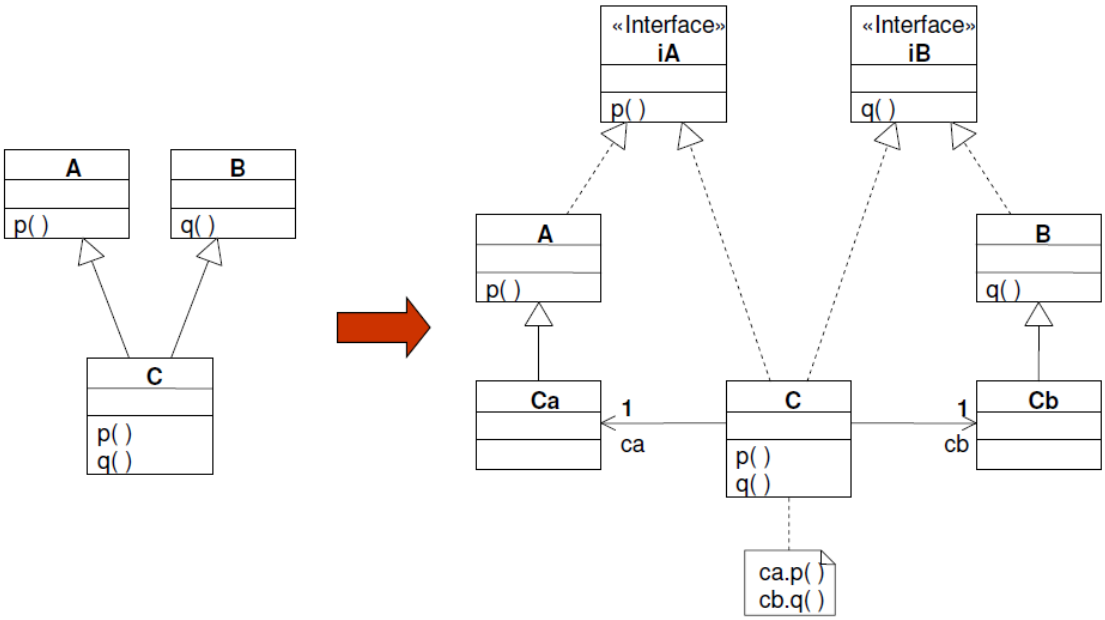

Ilustración 26: Conversión herencia [18]

Las clases que no estén involucrados en la multiherencia se dejarán igual que estaban, únicamente deberán ser copiadas.

Ahora se muestra como quedó el paquete Uno una vez fue utilizada la aplicación sobre el caso de prueba.

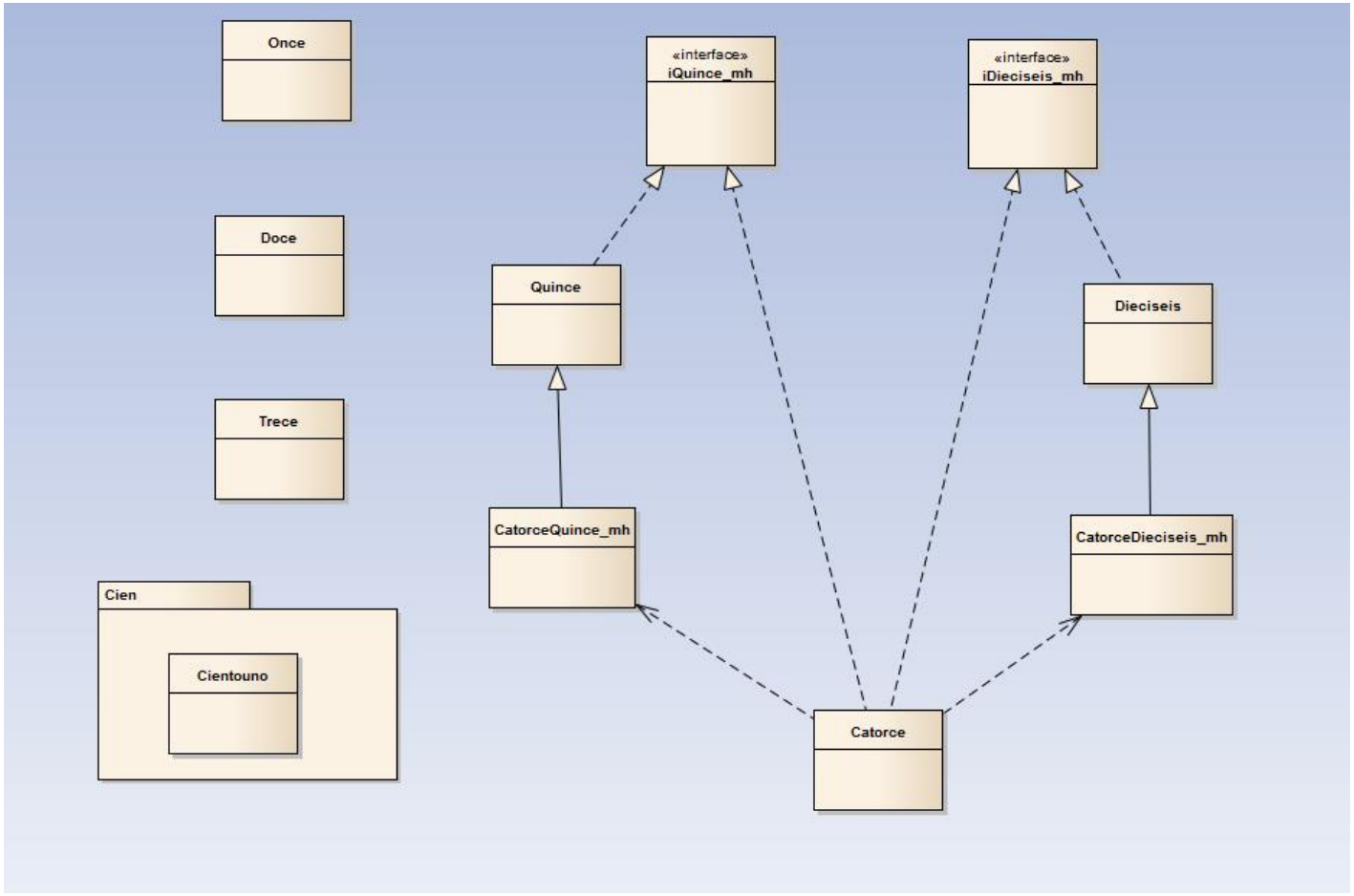

Ilustración 27: Conversión paquete Uno

Se puede ver cómo aquellas clases que no estaban afectadas por la herencia múltiple han seguido exactamente igual. Hablamos de clases como Once. Doce y Trece. Además el paquete Cien tampoco ha sido modificado, al igual que todo el contenido almacenado en él, en este caso la clase java Cientouno.

En cambio todas las clases relacionadas con aquella que implementaba la herencia múltiple, la clase Catorce, han sido modificadas de acuerdo a cómo se explicó anteriormente. Los distintos cambios aplicados son los siguientes:

- Se han añadido dos interfaces, iQuince_mh y iDieciseis_mh.
- Las clases Quince y Dieciséis se han modificado. Además, implementarán las interfaces mencionadas anteriormente.
- Se han creado las clases CatorceQuince_mh y CatorceDieciseis_mh. Además, estás clases pasarán a ser hijas de las clases Quince y Dieciséis respectivamente.
- La clase Catorce ha sido modificada. Ahora implementará además a las dos interfaces.

Con todo esto conseguimos el mismo efecto físico que teníamos antes, ya que previamente por herencia la clase Catorce tenía que tener todos los atributos y todos los métodos que sus clases padres, Quince y Dieciséis, tuviesen.

Ahora la clase Catorce no guarda ninguna relación con las otras dos clases, pero igualmente tendrá, y nos aseguramos de ello con esta implementación, todas las propiedades (atributos y métodos) que las clases que otrora fueran sus padres.

¿Esto por qué es? Esto se debe a que la interfaz que implementa la clase Catorce y que también implementa la que antes era su clase padre, tiene todos los métodos y atributos que tuviese su padre. Además, la clase "compuesta" (por ejemplo CatorceQuince_mh), al heredar de la clase padre, tendrá todas las propiedades del padre.

La clase hija, Catorce en este caso, ha sido modificada para contener todas las propiedades que la clase compuesta tiene. Esto se ha hecho a través de métodos implementados en la clase hija, donde únicamente se llama al método equivalente de la clase compuesta.

Por tanto, y tras hacerse por ambos lados (como es herencia múltiple significa que tiene más de un padre, se deberá hacer para todos por igual, claro), la clase hija tendrá todas las propiedades que tienen sus distintos padres, al igual que si tuviese multiherencia, pero sin vulnerar las especificaciones que Java nos ofrece.

La única excepción a esto son los métodos y atributos privados, que no se obtendrán. Pero esto es igual en la forma que tiene Java de trabajar con la herencia,

aquellos métodos y atributos privados solo pueden ser manejados en la clase en la cual se implementaron.

7.3.3 Prueba: paquete Dos

Aquí tenemos otro ejemplo. Como se comentó anteriormente, el programa recorre recursivamente todas las carpetas, y en este caso ha entrado en el último paquete de la aplicación de prueba, el paquete llamado Dos.

Se puede ver a continuación:

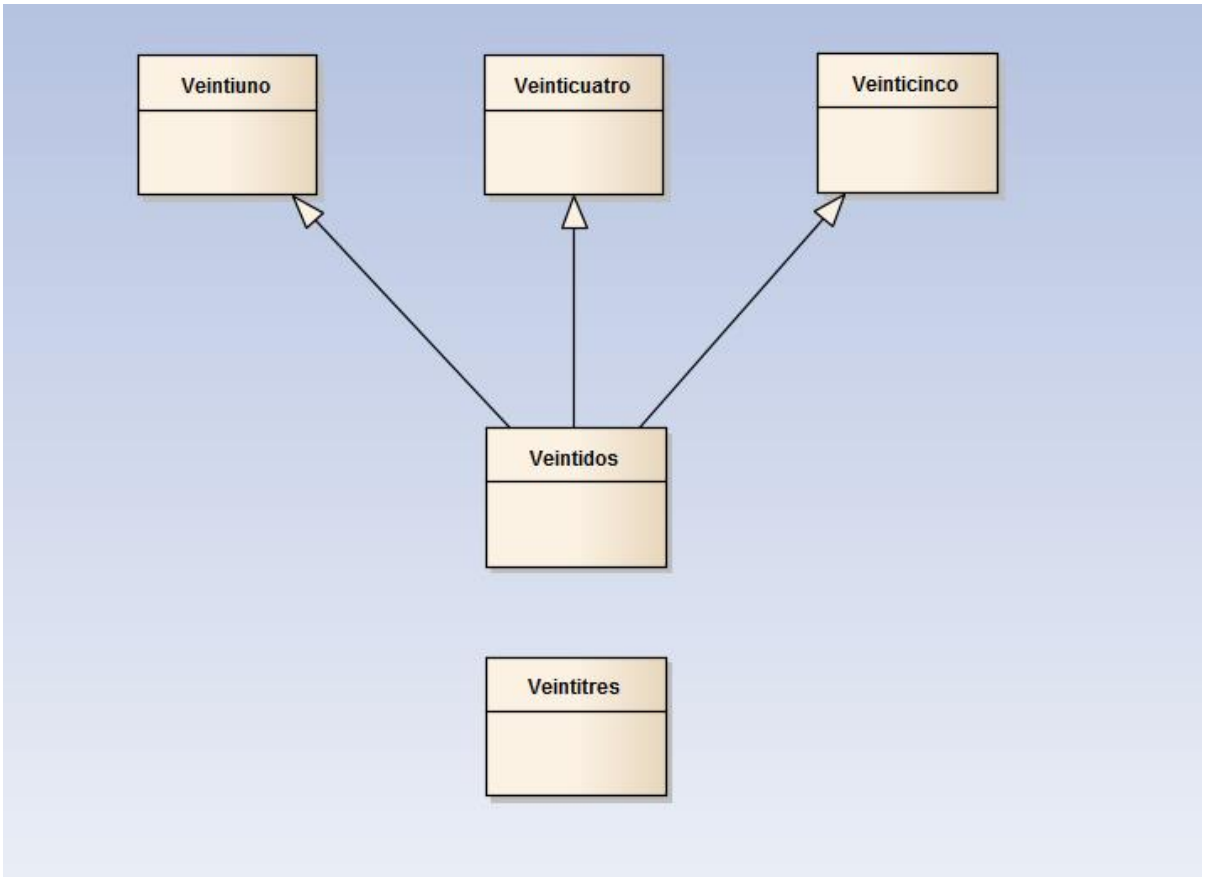

Ilustración 28: Paquete Dos

Vemos en este caso que el paquete tiene de nuevo herencia múltiple. Pero a diferencia del caso anterior, donde había dos padres, en este caso el número de padres es de tres.

Los diferentes elementos de este paquete son los siguientes:

- Existe una clase hija, Veintidós.
- Hay tres clases que harían la función de padre, Veintiuno, Veinticuatro y Veinticinco.
- Existe una clase que no está relacionada con la herencia múltiple, Veintitrés.

La conversión que se debe llevar a cabo es la misma que en el apartado anterior, siempre será la misma, de hecho.

Una vez pasado el programa a la aplicación de prueba, el resultado obtenido para este paquete, es el siguiente:

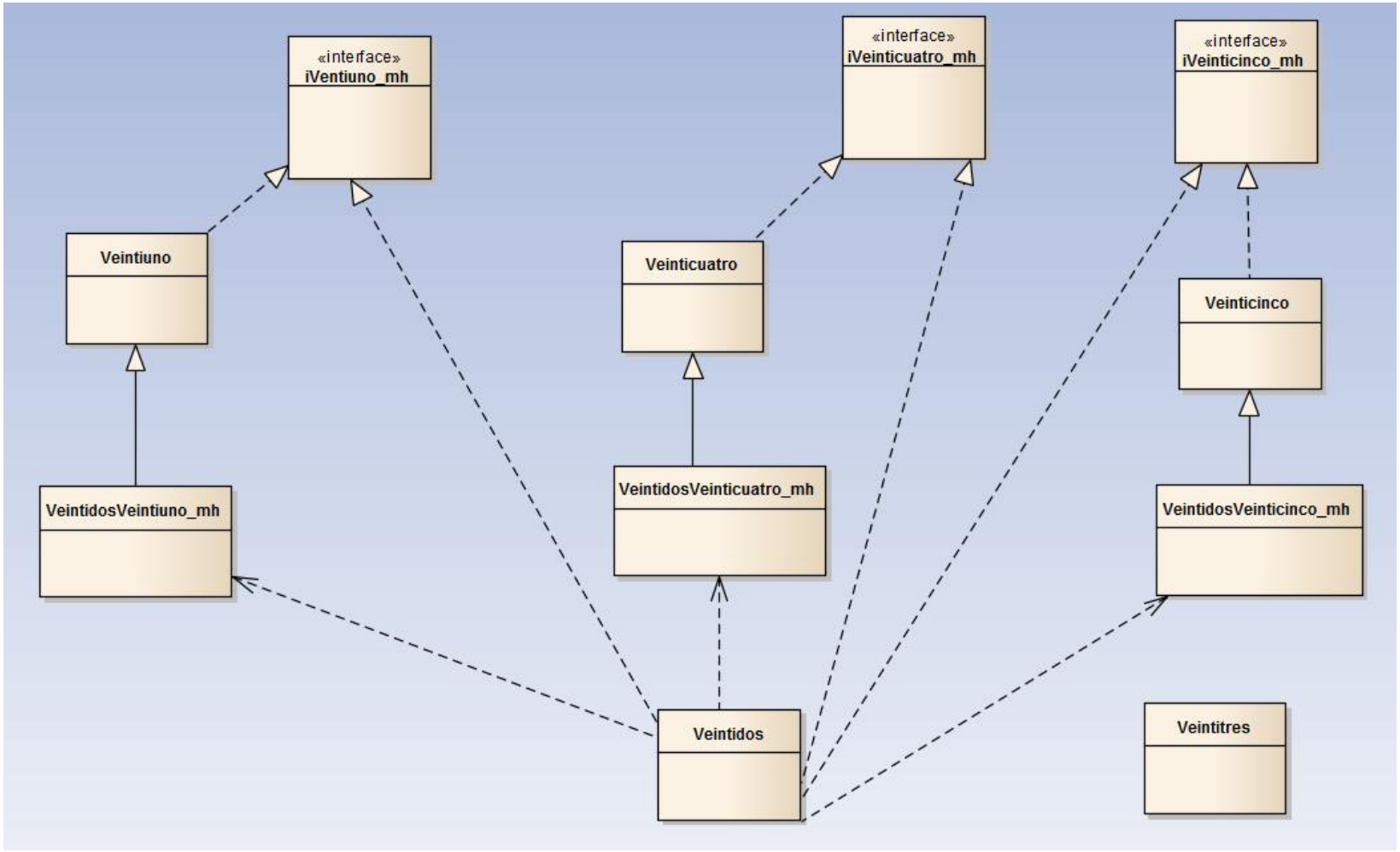

Ilustración 29: Conversión paquete Dos

Se pueden observar los cambios perfectamente. Por supuesto, sigue lo establecido en los puntos anteriores.

La clase Veintitrés, al no participar en la herencia múltiple, sigue exactamente igual. El resto de clases e interfaces si han sido añadidas o modificadas.

Se describen a continuación el número de elementos que han sido modificados/añadidos:

- Se han creado tres interfaces. Una por cada padre que había anteriormente en la herencia múltiple. Las interfaces son iVeintiuno_mh, iVeinticuatro_mh y iVeinticinco_mh.

- Las clases padre han sido modificadas, las tres. Además cada una pasará a implementar a la interfaz que le corresponda. Estas clases son Veintiuno, Veinticuatro y Veinticinco.

- Se han creado un total de tres clases Java nuevas, que son intermediarias entre las clases padre y la clase hija. Son las clases VeintidosVeintiuno_mh, VeintidosVeinticuatro_mh y VeintidosVeinticinco_mh. Además, estas clases serán ahora hijas de las clases padres mencionadas anteriormente, con herencia simple, claro.

- Además la clase hija, Veintidós, ha sido modificada. Ha dejado de pertenecer a ningún tipo de herencia, pero por el contrario ha pasado a implementar a las tres interfaces mencionadas anteriormente.

Al igual que en el caso anterior, la funcionalidad de la aplicación tras pasarse por ella el programa ha quedado igual. Además, los métodos y atributos que tendría la clase hija en caso de poder realizar la multiherencia serían los mismos que de esta manera obtiene. A priori no hay ningún tipo de diferencia funcional.

Podemos ver también que en la herencia múltiple no es dos el número máximo de padres que pueda haber, si no que el número máximo de padres que se puede tener ahora en un programa Java es infinito, aún a pesar de que Java te lo limite a uno.

7.4 Conclusiones

Se puede concluir que las pruebas sobre la aplicación han sido definidas previamente de una manera muy detalla, intentando que nada quedara fuera de lo probado.

Por supuesto esto es totalmente irreal, ya que la prueba total es imposible. Pero al menos se ha tratado cubrir con estas pruebas las máximas situaciones posibles.

Primero se definieron y contaron aquellas pruebas estáticas y dinámicas hechas en la aplicación. Pruebas estáticas sobre el diseño, sobre el código e incluso sobre el documento; y las pruebas dinámicas tales como las pruebas unitarias, de integración, del sistema, etc.

Posteriormente se pasó a detallar una prueba concreta. Esto se hizo así porque se consideró que este tipo de prueba era lo que más se asemejaba a una situación real.

La totalidad de las pruebas han sido realizadas sobre Java 7, que es la versión de Java sobre la cual se realizó la aplicación previamente. En el tiempo que se ha tardado en realizar la memoria una nueva versión de Java vio la luz, y para ver si la aplicación sigue funcionando de manera correcta sobre ella se ha realizado una única prueba sobre ese compilador, cogiendo para ello la prueba de sistema más completa de la que disponíamos.

Por supuesto, se hicieron multitud de pruebas que no ha sido posible detallar en el documento. No habría espacio para todas, además de que el propósito de este documento no es ese.

Pero, desde luego, el plan de pruebas de cualquier desarrollo debe ser detallado y amplio, ya que la fase de pruebas es una de las principales en importancia en cualquier programa.

8. Datos asociados al proyecto

8.1 Planificación

A continuación se muestra un diagrama de Gantt, donde aparece reflejado la planificación que se hizo, a priori, de los tiempos que iban a ser necesarios para desarrollar el proyecto, desde el primer paso al último.

La fecha que se tomó inicialmente como fin de proyecto fue el 16 de Mayo de 2014, aunque esta fecha es únicamente una aproximación.

Name	Begin date	End date
Estudio documentación	5/6/13	5/29/13
Análisis detallado del sistema	5/29/13	6/25/13
Definición de requisitos de software	6/26/13	7/17/13
Definición de arquitectura	7/11/13	7/26/13
Diseño detallado	7/29/13	8/30/13
Diseño de pruebas	8/12/13	8/30/13
Implementación del software	9/2/13	12/27/13
Ejecución de las pruebas	12/18/13	2/13/14
Desarrollo de la memoria	1/20/14	4/25/14
Desarrollo del manual de usuario	4/8/14	4/21/14
Entrega del prototipo	5/16/14	5/16/14

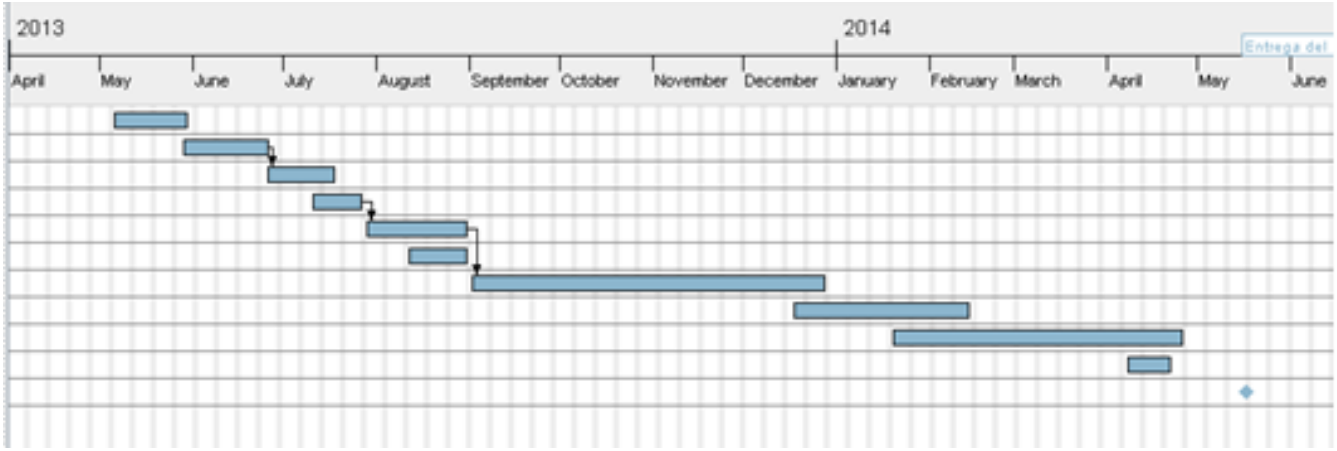

Ilustración 30: Planificación. Diagrama de Gantt.

El diagrama cuenta con once actividades, que se pueden separar en cinco grandes grupos:

- **Fase inicial:** incluye el estudio de la documentación y el análisis detallado del sistema.

- **Fase de análisis:** con la definición de requisitos de software y la definición de la arquitectura.

- **Fase de diseño:** que engloba al diseño detallado y al diseño de las pruebas.

- **Fase de implementación:** que además de la implementación del sistema añade la ejecución de las pruebas.

- **Fase de documentación:** incluye el desarrollo de la memoria y la elaboración del manual del usuario.

- Adicionalmente se incluye un punto, un milestone final, que es la entrega del prototipo final.

A continuación se procede a explicar brevemente las distintas actividades:

- **Estudio de la documentación:** se trata a la lectura y análisis de la documentación entregada por el tutor.

- **Análisis detallado del sistema:** es una ampliación al punto anterior, añadiendo distintas reuniones que se tuvieron con el tutor al principio del proyecto, para puntualizar los distintos apartados del programa, así como el conocimiento añadido por parte de las búsquedas, tanto en internet como en distinta bibliografía.

- **Definición de requisitos:** una vez se tiene todo claro, se procede a definir los requisitos. Esta parte se reutilizará posteriormente, añadiéndolo en la memoria.

- **Definición de la arquitectura:** se trata de un grupo de decisiones técnicas que deben ser tomadas. Lenguaje de programación, bases de datos a utilizar, si necesita de algún apoyo, ya sea un servidor o una máquina virtual, etc. En este caso fue sencillo y la labor se llevó a cabo en muy poco tiempo.

- **Diseño detallado**: incluye diagramas de casos de uso, diagramas de clase, etc. De nuevo, es reutilizable para la memoria.

- **Diseño de las pruebas:** se procede a discernir qué pruebas se realizarán sobre la aplicación una vez esté desarrollada.

- **Implementación del sistema:** es la fase más larga. Consiste en la realización del código de la aplicación.

- **Ejecución de las pruebas:** se prueba la aplicación que ha sido desarrollada en el punto anterior. Se realizarán las pruebas diseñada anteriormente. Además, durante la fase de implementación se irán haciendo pequeñas pruebas, que no aparecerán reflejadas en este apartado.

- **Desarrollo de la memoria:** se escribirá un Word, este concretamente, con la memoria de la aplicación.

- **Elaboración del manual de usuario:** se creará otro Word donde se explica al usuario cómo ejecutar la aplicación.

8.2 Herramientas utilizadas

A continuación se explicará de manera ligera los programas que han sido utilizados para la realización de este proyecto.

Para ver el coste económico de dicho software habría que consultar el punto siguiente, en el que se hace mención al presupuesto de software.

El software utilizado ha sido el siguiente:

- Enterprise Architect
- Microsoft Office 2010
- Eclipse
- GanttProject
- EditPlus

Enterprise Architect

Se trata de un programa que sirve para modelar, diseñar y visualizar software. Su principal misión es el diseño de software a través de UML 2.1. Existen dos versiones, una gratuita y una de pago, que incluye funcionalidades extras, como la exportación del diseño. [20]

Ha sido utilizado para el diseño de la aplicación.

Ilustración 31: Enterprise Architect [21]

Microsoft Office 2010

El archiconocido Microsoft Office es un paquete de ofimática de Microsoft que incluye diversos programas. Entre ellos destacan el Microsoft Word, para redactar documentos, el Microsoft Excel, con el que se pueden crear hojas de cálculo y el PowerPoint, con el que se pueden diseñar presentaciones. Además existen otros programas, tales como el Outlook (correo) o el Access (base de datos).

La versión utilizada es la 2010, que sucede a la 2007. Se trata de una versión más ligera y trae alguna pequeña modificación, pero la esencia es la misma.

Microsoft Office 2010 soporta los siguientes formatos:

DOC, DOCX, XLS, XLSX, PPT, PPTX, MDB, ACCDB, PUB, RTF, TXT, HTM, JPG, PNG, TIF, EMF, WMF, XML, WRI, ODT, ODP, ODS, WMV, AVI, PDF [22]

Para este proyecto han sido utilizados únicamente dos herramientas del conjunto que tiene Microsoft Office 2010, Microsoft Word, para la creación de la memoria, y Microsoft PowerPoint, para la creación de la presentación.

Ilustración 32: Microsoft Office [23]

Eclipse Juno [24]

Eclipse es el programa que se ha utilizado para desarrollar el código fuente. Se trata de un programa libre, que se puede descargar gratuitamente en internet, bien con el compilador incluido o sin él. Actualmente está desarrollado por la Fundación Eclipse, aunque en un principio pertenecía a IBM.

Eclipse se trata de un IDE (Integrate Development Enviroment), esto es, un entorno que permite a un programador desarrollar una aplicación. Eclipse acepta varios lenguajes de programación, aunque el más común es Java.

Para programar en este lenguaje existen otros IDE, tales como NetBeans [25], BlueJ [26] o JBuilder [27].

Además Eclipse tiene multitud de plugins, que son pequeños programas que puedes añadir al Eclipse original que te dan mucha más potencia, añadiéndole más funcionalidades al IDE. Esto te permite integrar funcionalidades que antes hacían programas externos en el propio Eclipse.

Ha sido utilizado para el desarrollo de código. Existen multitud de versiones de Eclipse, la mayoría de ellas bastante parecidas. En este caso ha sido utilizada la versión Juno.

Ilustración 33: Eclipse Juno [28]

GanttProject [29]

Se trata de una herramienta muy útil para poder representar gráficamente la planificación de un proyecto y hacer un seguimiento posteriormente del mismo. Es totalmente gratutita.

Con GanttProject el control que se obtiene sobre el proyecto es casi total. Puedes hacer una planificación general e ir independizándola por recurso, dividiendo las distintas tareas principales en otras más pequeñas que podrán ser asignadas a los distintos recursos del proyecto. Además permite crear dependencias entre tareas.

Adicionalmente se podrá exportar la planificación del proyecto a distintos formatos como PDF, HTML o a una imagen (JPG, PNG).

Ha sido utilizado para realizar la planificación del proyecto y su posterior seguimiento.

Ilustración 34: Gantt Project [30]

EditPlus

EditPlus es uno de los editores de texto más utilizados actualmente. Actualmente conocido como Edit++, se utiliza en Windows.

Se trata de una mejora sobre el bloc de notas que tiene incorporado el propio Windows, ya que tiene varias funciones donde destaca el resalto de errores sintácticos sobre ciertos lenguajes (destacan Java, C, C++, HTML, PHP y JavaScript), pero sigue siendo un editor ligero y fluido.

EditPlus ha sido utilizado para crear, ver y modificar los ficheros de pruebas utilizados en esta aplicación.

Ilustración 35: Edit Plus [31]

9. Presupuesto

En este apartado se procederá a detallar el presupuesto estimado acerca del coste derivado del proyecto en caso de haber sido realizado en un entorno profesional. La plantilla para calcular dicho presupuesto ha sido obtenida que ha sido utilizada es aquella que la universidad Carlos III pone a disposición de los alumnos.

En ningún caso se pretende establecer un precio que un cliente debería pagar por este producto, ya que, entre otras cosas, no se van a considerar márgenes de riesgo o beneficio.

9.1 Costes de personal

Con respecto al coste de personal ha habido un único desarrollador, que ha realizado los distintos roles que ha requerido la aplicación a través de las distintas fases:

- **Analista**: en la fase toma de requisitos.
- **Diseñador**: en la fase de diseño.
- **Programador**: en la fase de implementación.
- **Testeador**: en la fase de pruebas.

El coste de personal ha sido tomado en hombre/mes. Una unidad de hombre/mes es el precio estimado de un trabajador por mes de trabajo. El precio de hombre/mes estimado ha sido tomado de la plantilla de presupuesto puesta a disposición de los alumnos por la universidad Carlos III de Madrid.

El tiempo en el cual ha sido realizado el proyecto es de diez meses, pero como la dedicación del desarrollador ha sido parcial, el número total de hombres/mes a computar es de la mitad, cinco.

La tabla de coste personal quedaría como sigue:

Nombre y apellidos	Categoría	Dedicación *	Coste *	Coste total
Miguel Alonso Luna	Ingeniero Informático	5 meses	2.694,39 €	**13.471,95 €**

Tabla 24: Costes de personal

* Medidos en hombre/hora

9.2 Costes de hardware y software

En este punto se procederá a detallar el hardware y software que ha sido necesario para desarrollar el producto.

Nuevamente los datos utilizados, tales como el período de amortización, han sido obtenidos de la plantilla que la universidad Carlos III proporciona a sus alumnos.

Para el cálculo del coste imputable de cada uno de los elementos hardware y software se hace uso de la siguiente tabla:

$$\frac{A}{B} \; x \; C \; x \; D$$

Donde,

A es el número de meses de uso del material.

B es el periodo de amortización.

C es el coste del equipo antes del IVA.

D es el porcentaje de uso que se dedica al proyecto.

Descripción	Coste	% Uso	Dedicación (meses)	Período de amortización	Coste imputable
Ordenador de sobremesa	700 €	100%	10	60	116,67 €
Microsoft Windows 7	120 €	100%	10	60	20 €
Microsoft Office 2010	235 €	100%	10	60	39,17 €
Enterprise Architect	120 €	100%	10	60	20 €
Eclipse	0 €	100%	10	60	0 €
GanttProject	0 €	100%	10	60	0 €
EditPlus	0 €	100%	10	60	0 €
				Total	**195,84 €**

Tabla 25: Costes de hardware y software

9.3 Otros costes

No se han incluido otros gastos como dietas o transportes ya que no han sido necesarios para la realización del proyecto.

9.4 Resumen de costes

A continuación se desglosan los gastos antes de IVA:

Concepto	Coste
Personal	13.471,95 €
Hardware y software	195,84 €
Total	**13.667,79 €**

Tabla 26: Resumen de costes

El coste total del proyecto, sin haber aplicado IVA, asciende a **TRECE MIL SEISCIENTOS SESENTA Y SIETE CON SETENTA Y NUEVE** euros.

El presupuesto total añadiendo el IVA asciende a:

Concepto	Coste
Personal	13.471,95 €
Hardware y software	195,84 €
Costes indirectos (IVA 21%)	2.870,24 €
Total	**16.538,03 €**

Tabla 27: Presupuesto final con IVA

El coste total del proyecto ya con el IVA aplicado ascendería a **DIECISÉIS MIL QUINIENTOS TREINTA Y OCHO EUROS CON TRES CÉNTIMOS.**

10. Conclusiones y trabajos futuros

10.1 Conclusiones

La finalización del proyecto me genera mucha satisfacción, desde diversos puntos de vista. Técnicamente me aporta una satisfacción enorme por el hecho de haber acabado con buen pie finalmente este proyecto, pero es que además el finalizar el proyecto supone además poner punto y final a mis estudios universitarios.

Académicamente para mi persona ha sido muy enriquecedor, pues aunque ya conocía en bastante profundidad el lenguaje de programación Java, me ha permitido aun así obtener un gran número de conocimientos nuevos.

Entre ellos destaco el haber tenido la posibilidad de haber realizado un proyecto desde cero, pasando por todas las fases del mismo. Ha sido muy satisfactorio el haber podido empezar en el diseño, y es más, incluso en la discusión de qué y cómo iba a ser el proyecto, llegando al fin hasta las pruebas y la documentación final, pasando por todos los estados intermedios, como la programación, por supuesto.

Además me ha servido para profundizar mis conocimientos en materias que si bien conocía, desde luego no lo hacía al nivel que ahora lo hago. Puedo incluir en este punto, por ejemplo, todo el diseño de Java, como afecta la herencia al programa, los modificadores de acceso, etc. Creo que esto me puede ser de gran utilidad en el futuro, porque he alcanzado un grado de conocimiento de estos asuntos mayor.

Respecto a la aplicación, pese a estar finalizada en este punto, permite futuras actualizaciones o mejoras, pudiendo hacer valiosas incorporaciones en ella en el futuro, pese a estar actualmente totalmente operativa y funcional.

10.2 Dificultades encontradas

Al tratarse de un proyecto que comenzó de cero se ha dado lugar a varias dificultades para realizar la aplicación. La más clara es que, en un comienzo, esta aplicación es simplemente una idea, un concepto. No hay nada realizado similar, luego hay mucho, y cuando digo mucho realmente es muchísimo, que pensar, multitud de cosas que no se te ocurren hasta que estás inmerso en ellas.

Se trata esto de una gran dificultad del proyecto, aunque al mismo tiempo es un gran reto, y hace, de verdad, que la realización de este proyecto haya sido más entretenida, divertido incluso por momentos, y encima obteniendo la satisfacción de haber sido tú quien ha pensado prácticamente todo lo que aquí se ha hecho.

Esto es debido a que no solo el proyecto se comenzó de cero, sino que además no se encontró nada similar. Es un proyecto totalmente nuevo. Sí es posible que algo parecido se haya planteado, e incluso realizado previamente, pero yo desde luego no lo he encontrado.

Otro problema encontrado en la realización de la aplicación principalmente, es que el programa acepta todo. Con esto nos referimos a que modifica el código de alguien, y debe aceptar todo lo que alguien haga.

Esto es, la entrada que la aplicación recibe es, en esencia, cualquiera. Esto complica sobremanera el desarrollo ya que debe prepararse el programa para funcionar ante cualquier cosa, y esto siempre es enormemente complicado.

10.3 Trabajos futuros

El programa trata totalmente el problema de la herencia múltiple en Java. Es un problema concluso. Lo cual no significa que no se pueda seguir trabajando en él para añadirle nuevas propiedades o integrarlo en otras soluciones.

Por ejemplo, en el punto anterior se comentaba que la entrada de la aplicación es cualquiera que nuestra mente pueda imaginar. Cualquier código que Java permite la aplicación la acepta. Y Java permite mucho. Muchas posibilidades se probaron también,

pero estoy convencido que siempre hay algo nuevo que se puede introducir y que ni al más imaginativo se le ocurrió probar. Siempre se puede probar más, todo.

Además, Java se va actualizando. Esta aplicación ha sido desarrollada usando una versión de Java 7. Me refiero, por supuesto, a la versión de Java que utiliza el programa de entrada, el que la aplicación cambiará si la herencia múltiple es encontrada. En el momento de la realización del código del proyecto Java estaba actualizando la versión a la 8 [32]. Y en el futuro sacará nuevas versiones.

Normalmente estas modificaciones que va sacando Java no afectan al programa. De hecho se ha realizado una prueba con Java 8 y la aplicación funciona correctamente, pero desde luego creo que hay que seguir vigilando las actualizaciones Java en el futuro por si algún cambio sí pudiese afectar. Y probando los programas de entrada, utilizando nuevos programas por cada versión con sus cambios, desde luego. Al menos los cambios que pueden afectar a aquello en lo que se fija esta aplicación (manera de definir los parámetros, accesos, etc.)

Y luego está la interfaz gráfica. En esta aplicación es muy básica, por comandos, desde luego se puede cambiar. Aunque personalmente creo que no es necesario, ya que este programa no busca eso, alguien quizás podría opinar lo contrario.

Pero, desde luego, la mejora grande y enormemente positiva de cara al futuro en esta aplicación, en lugar de cambiar la interfaz gráfica, sería eliminar cualquier interfaz de usuario y en su lugar integrar esta aplicación en los programas de desarrollo.

Una gran mejora sería, por ejemplo, añadir un plugin en el Eclipse. De esta manera, simplemente presionando un botón, el editor buscaría las distintas herencias múltiples y crearía el nuevo programa con la modificación creada a través del uso de esta aplicación, en la carpeta que nosotros hayamos definido previamente.

Bibliografía

[1]. Eckel, Bruce. 2000. Thinking in Java. Prentice Hall.

[2]. Stroustrup, B. 2000. The C++ Programming Language. Addison Wesley.

[3]. Oscar León, Mariana Brachetta, Julio Monetti. Herencia múltiple en Java. Universidad Tecnológica de Mendoza. Páginas 6-13.

[4]. Compilador (Febrero 2014): http://www.carlospes.com/minidiccionario/compilador.php

[5]. Fases del compilador (Febrero 2014): http://www.monografias.com/trabajos11/compil/compil.shtml

[6]. Herencia simple (Marzo 2014): http://profesores.fi-b.unam.mx/carlos/java/java_basico3_4.html

[7]. Atributos y métodos en herencia (Marzo 2014): http://www.unav.es/SI/manuales/Java/indice.html

[8]. Interfaces. Ejemplo código (Febrero 2014): http://dis.um.es/~bmoros/Tutorial/parte5/cap5-12.html

[9]. Interfaces Java (Febrero 2014): http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=671:para-que-sirven-las-interfaces-java-implementar-una-interfaz-del-api-ventajas-y-ejemplos-basicos-cu00697b&catid=68:curso-aprender-programacion-java-desde-cero&Itemid=188

[10]. Gonzalo Génova. Proyecto práctico de diseño de software. Universidad Carlos III. Página 140.

[11].Gonzalo Génova. Proyecto práctico de diseño de software. Universidad Carlos III. Página 142.

[12].Gonzalo Génova. Proyecto práctico de diseño de software. Universidad Carlos III. Página 143.

[13].Cristina Cachero, Pedro J. Ponce de León. Programación orientada a objetos, tema 3: Herencia. Universidad de Alicante. Páginas 52-54.

[14].Ghan Bir Singh. Single Versus Multiple Inheritance in Object Oriented Programming. Rensselaer Polytechnic Institute at HGC, Hartford, CT, USA. Páginas 1-2 y 9-10.

[15].Herencia (Marzo 2014): http://www.zator.com/Cpp/E4_11_2c.htm

[16].Oscar León, Mariana Brachetta, Julio Monetti. Herencia múltiple en Java. Universidad Tecnológica de Mendoza.

[17].Jorge Luis Díaz Suarez. Curso OO: "La herencia, más aumento de la ambigüedad". Páginas 128-130.

[18].Java volumetría (Abril 2014): http://www.visualbeta.es/28186/general/java-es-el-lenguaje-mas-utilizado-por-los-desarrolladores/

[19].Gonzalo Génova. Proyecto práctico de diseño de software. Universidad Carlos III. Página 144.

[20].Enterprise Architect (Marzo 2014): http://www.sparxsystems.es/

[21].Enterprise Architect logo (Marzo 2014): http://www.logotypes101.com/free_vector_logo/96823/Enterprise_Architect

[22]. Microsoft Office 2010 (Marzo 2014): http://microsoft-office-2010.softonic.com/

[23].Microsoft Office 2010 logo (Marzo 2014): http://qrtecnologia.com/microsoft-lanza-office-online/

[24].Eclipse Juno (Marzo 2014): http://www.eclipse.org/juno/

[25].NetBeans (Marzo 2014): http://www.netbeans.org

[26].BlueJ (Marzo 2014): http://bluej.org

[27].JBuilder (Marzo 2014): http://www.embarcadero.com

[28].Eclipse Juno logo (Marzo 2014): http://www.eclipse.org/eclipselink/releases/2.4.php

[29].Gantt Project (Marzo 2014): http://ganttproject.softonic.com/

[30].Gantt Project logo (Marzo 2014): https://ppguillemcuberta.wordpress.com/tag/ganttproject/

[31].Edit Plus (Marzo 2014): http://www.e-coffeetech.com/articulos/desarrollo-de-software/106-editplus-editor-de-texto-para-programadores.html

[32]. Java Versión 8 (Noviembre 2014): http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html

[33]. Bucle, definición (Marzo 2014): http://es.wikipedia.org/wiki/Bucle_(programaci%C3%B3n)

[34].Clase (Abril 2014): http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=411:conceptos-de-objetos-y-clases-en-java-definicion-de-instancia-ejemplos-basicos-y-practicos-cu00619b&catid=68:curso-aprender-programacion-java-desde-cero&Itemid=188

[35]. Herencia (Marzo 2014): http://profesores.fi-b.unam.mx/carlos/java/java_basico3_4.html

[36]. UML (Abril 2014): www.uml.org

Anexo A: Glosario

- **Algoritmo:** Es un conjunto de instrucciones prescritas que permite llegar desde un estado inicial a uno final que implicaría la solución.
- **Atributo:** Propiedad que tiene la clase o el objeto.
- **Bucle:** Es una sentencia que se realiza repetidas veces a un trozo aislado de código, hasta que la condición asignada a dicho bucle deje de cumplirse. [33]
- **Clase:** Abstracción que define un tipo de objeto especificando qué propiedades (atributos) y operaciones (métodos) disponibles va a tener. [34]
- **Desarrollador:** Persona encargada de la implementación del software.
- **Herencia:** La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. [35]
- **Herencia múltiple:** tipo de herencia donde en lugar de existir una única superclase se tienen dos o más.
- **Herramientas case:** herramientas para el desarrollo informático que tienen como principal motivo ayudar al desarrollador para que reduzca sus tiempos y aumente, por tanto, su productividad. Entre sus características destaca la obtención de código fuente automáticamente desde el diagrama de clases.
- **IDE de desarrollo:** entorno de desarrollo integrado. Es un programa informático compuesto por un conjunto de herramientas de programación.
- **Interfaz de usuario:** Se trata de un entorno visual sencillo que permite la comunicación entre el usuario y la aplicación.
- **Método:** Conjunto de sentencias que realizan una acción en la clase.

- **Migración:** Conjunto de actuaciones consistentes en la sustitución de una aplicación por otra distinta pero con funciones equivalentes.
- **MVC o Modelo-Vista-Controlador:** Arquitectura software en tres capas, donde habitualmente el Modelo interactúa con la bases de datos, la parte vista con el usuario y el controlador es el manejador de la aplicación.
- **Paquete:** Se trata de un contenedor de clases que permite agrupar las distintas partes de un programa cuya funcionalidad tienen elementos comunes.
- **Patrón:** Conjunto de soluciones ya diseñadas para solventar problemas comunes en el desarrollo de software.
- **Recursividad:** Se trata de un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo.
- **Subclase o clase hija:** En herencia, clase que hereda.
- **Sufijo:** Añadido que se hace al final de los nombres de la clase, para saber que han sido modificadas.
- **Superclase o clase padre:** En herencia, clase de la cual se hereda.
- **UML:** Unified Modeling Language [36]
- **Usuario:** Persona que utilizará la aplicación.

Anexo B: Manual de Usuario

Introducción

A continuación se explica brevemente como ejecutar la aplicación. Hay varias maneras de ejecutarse, ya que es un proyecto Java, pero en este caso se explica únicamente su ejecución usando la herramienta Eclipse.

De todas formas, si no se ejecuta por Eclipse y se hace de alguna otra manera (en la consola por líneas de comandos, usando NetBeans, etc.) la ejecución de la misma sería similar.

Se optó por esta modalidad de ejecución, con Eclipse, ya que este programa está diseñado especialmente para desarrolladores Java, y Eclipse es la herramienta más común entre los programadores de este lenguaje.

Así mismo el manual al estar pensado para programadores puede que se salte algún paso previo que gente que no desarrolle quizás no conozca, como puede ser la instalación del Eclipse. Pero como partimos de la base de quien va a ejecutar este programa desarrolla en Java, sabemos que habrá instalado el Eclipse previamente, seguramente unas cuantas veces, de hecho.

Igualmente la aplicación es muy fácil de usar, es una de sus virtudes, su sencillez de uso. Por eso el manual será muy sencillo y muy breve.

Ejecución

Como se puede ver en la imagen inferior, lo primero que se debe hacer, una vez tengamos el Eclipse abierto, es ejecutar la aplicación. Para ello se debe presionar en el botón señalado por el recuadro rojo.

Una vez hecho esto se seleccionará el programa que queremos ejecutar, Interfaz en este caso.

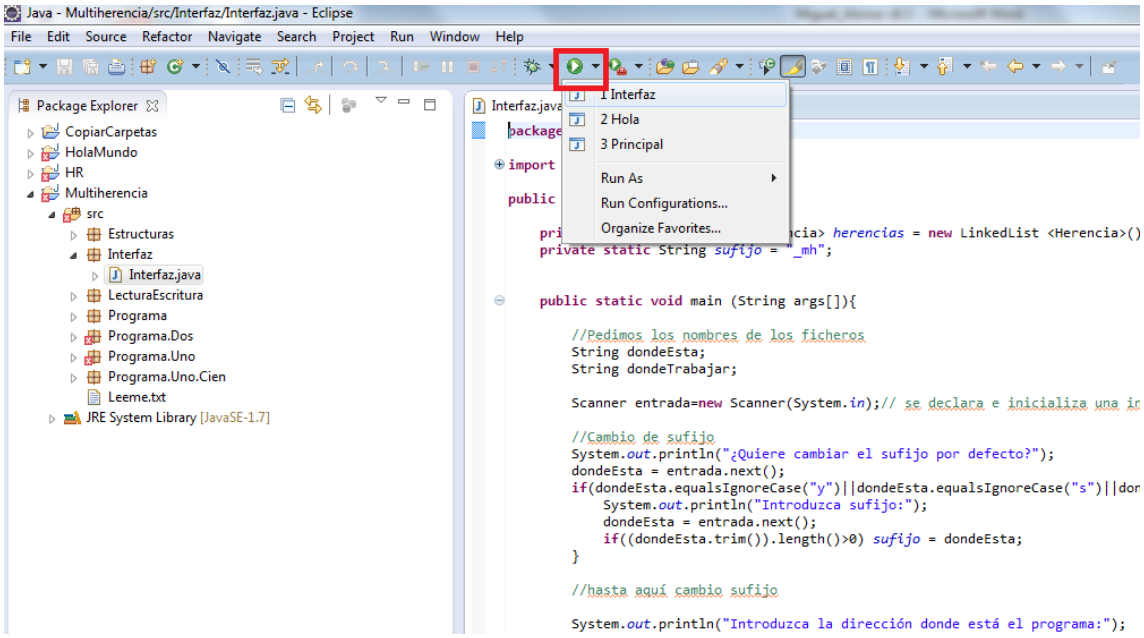

Ilustración 36: Ejecución con Eclipse

Una vez hecho esto en la consola de comandos (por defecto en la parte inferior del Eclipse) nos preguntará si queremos cambiar el sufijo. Podemos responder tanto en español como en inglés, y también con las iniciales tan solo.

En caso de seleccionar que sí queremos un sufijo nuevo nos pedirá a continuación cual es el nuevo sufijo que le vamos a dar. Si se selecciona que no el sufijo por defecto es "_mh".

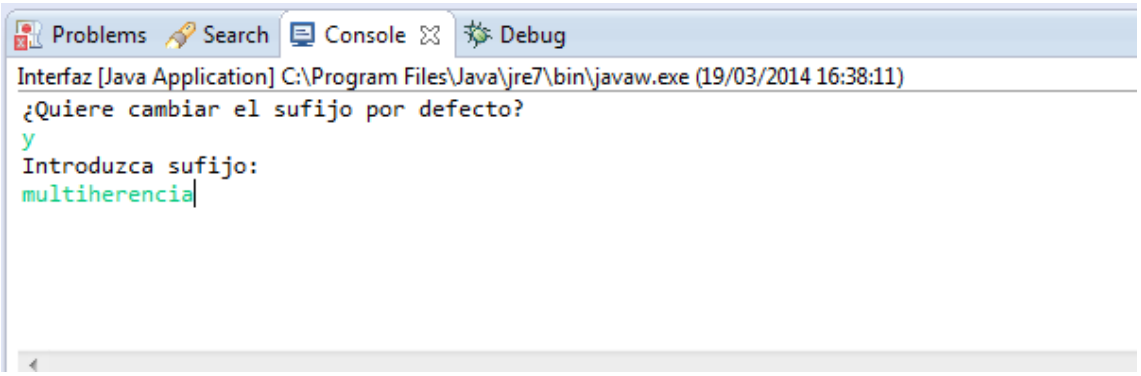

Ilustración 37: Selección sufijo

Una vez hecho esto se tendrán que decir las carpetas con las cuales trabajamos. La primera dirección que tenemos que escribir es aquella en la que está el programa, y la segunda es aquella donde el programa se copiará y donde será modificado.

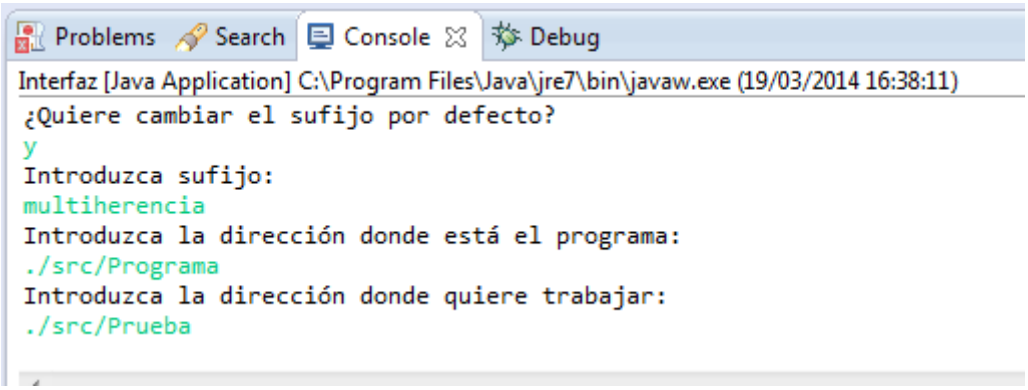

Ilustración 38: Selección carpetas

Una vez introducido esto el programa ejecutará solo, dando un mensaje de finalización.

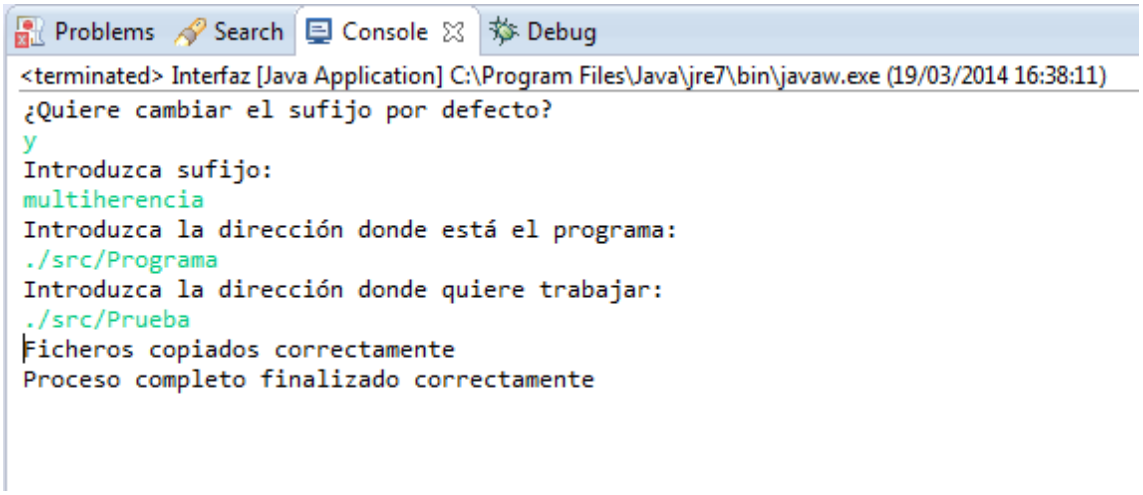

Ilustración 39: Finalización de ejecución

Excepciones

Existen casos en los que introducimos algún dato de forma incorrecta. En esos casos la aplicación nos avisa de esos errores.

Vemos algún ejemplo. En caso de introducir la dirección de entrada mal, daría lugar a que la aplicación no pudiese encontrar el programa que debiese modificar. Avisaría, en este caso, de la siguiente manera.

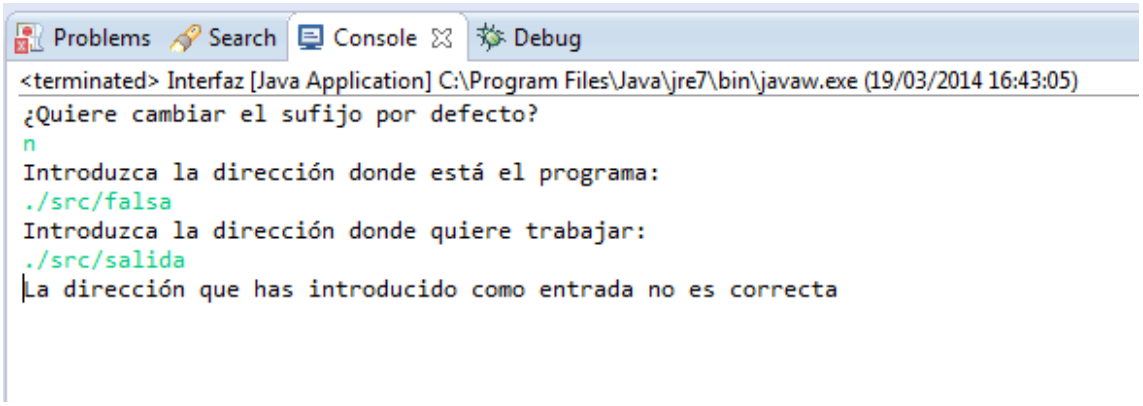

Ilustración 40: Ejemplo dirección de entrada incorrecta

En caso de introducir la dirección de salida mal (la dirección existe ya, no puede sobreescribir lo que en ella hay lógicamente), la aplicación mostraría el siguiente mensaje.

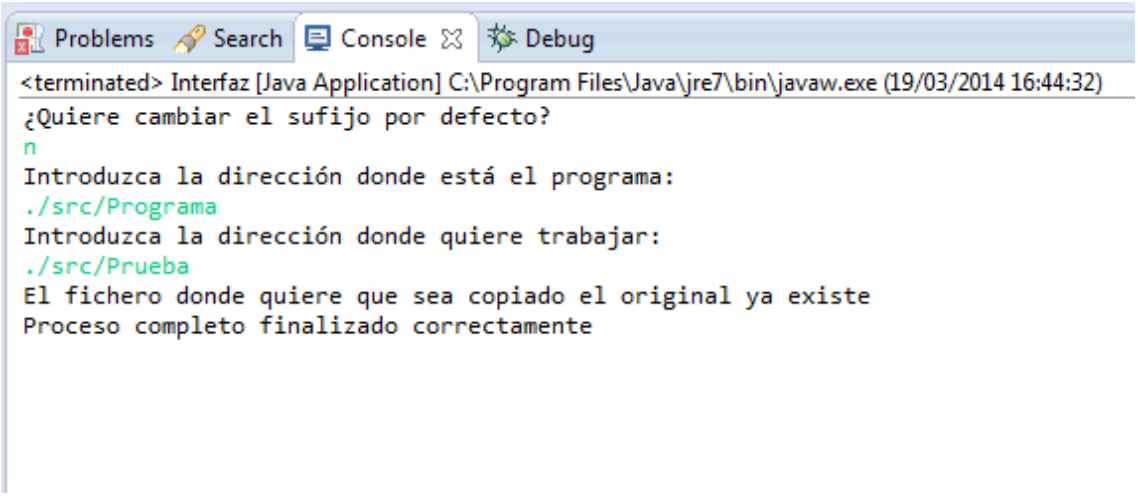

Ilustración 41: Ejemplo dirección de salida incorrecta